Escuela Politécnica Superior

Desarrollo de un videojuego para móviles con Unity o Cocos2d-x

Máster Universitario en Desarrollo de Software para Dispositivos Móviles

Trabajo Fin de Máster

Autor:
José María Egea Canales

Tutor/es:
Miguel Ángel Lozano Ortega

Septiembre 2016

Universitat d'Alacant
Universidad de Alicante

Justificación y Objetivos

Con el fin de poder aplicar los conceptos sobre videojuegos aprendidos en esta titulación de posgrado, y lograr tratar con los temas sociales y servicios de la plataforma Google, se pretende desarrollar un videojuego 2D sencillo pero completo para Android, que haga uso de servicios de Google, y que nos permita analizar y evaluar la dificultad con la que es posible trabajar con estos.

Para el desarrollo del videojuego se pretende utilizar el motor Cocos2d-x. Cabe recalcar que para la construcción correcta del proyecto, se realiza tanto el Documento de Diseño del Videojuego (GDD), como su posterior desarrollo utilizando el motor.

El videojuego estará realizado utilizando las bibliotecas de Cocos2d-x de C++, y se utilizará en la medida de lo posible las herramientas de este motor. Se pretende mantener un desarrollo limpio abierto a posibles mejoras y adaptación a dispositivos iOS en el futuro.

Como extra, para otorgarle más personalidad, se pretende realizar gráficos propios para todo el videojuego.

Agradecimientos

Este proyecto surge de mis problemas para prestar atención en ocasiones en clase, que me hacen acabar convirtiendo mis hojas de apuntes en hojas de garabatos, así que me agradezco a mí mismo por haber sacado algo productivo del desastre. Gracias, Yo.

Por otro lado, agradezco a mi compañero Juan Pomares, porque si lo pillabas de buenas, te resolvía todas las dudas que te pudieran surgir. Gracias, Juan, eres el king.

Finalmente, a mi tutor, Miguel Ángel, porque se apiadó de mi alma y no me ha presionado mucho a pesar de que mis procrastinaciones estaban a la orden el día, y otros asuntos más peliagudos. Gracias, MA.

Dedicatoria

Este trabajo se lo dedico a cualquier persona que esté ahora mismo leyendo estas líneas. Quizá ahora mismo acabas de madrugar en un lunes, llueve, y encima hace calor y viento, o quizá hoy es viernes y estás con los mejores planes que podrías haber hecho en meses, independientemente de todo esto, esta memoria va dedicada a ti, te lo mereces.

Citas

"Walking is not gameplay" – Scott Rogers

"Waka-Waka" – Pac Man

"Fin de la Cita" – Mariano Rajoy

"Os habéis esforzado, ¿y para qué? Para nada. Moraleja: No os esforcéis más" – Homer Simpson

Índice

1 Introducción

Con el paso del tiempo, las personas buscan nuevas formas de innovar, nuevas formas de entretenerse, de relacionarse, vivimos en un continuo cambio con foco en las nuevas tecnologías. Dentro de esto, mencionemos a los videojuegos, que con más de 70 años de historia[1], han evolucionado desde su aparición en máquinas de aplicación científica que con algo de ingenio se las modificaba para jugar al ajedrez o el tenis, a lo que son ahora, considerado como un arte por muchos, y máquinas de hacer dinero para otros, puesto que llegan a facturar anualmente más que el cine y la música juntos[2].

Además, en la actualidad, gracias al avance de la tecnología, los videojuegos han ampliado sus fronteras, no es raro ver videojuegos que aplican realidad aumentada y que son un éxito rotundo, como el reciente Pokémon Go, y que ha catapultado a Nintendo numerosos escalones más hacia el éxito[3], también se puede apreciar cómo empiezan los trasteos con la realidad virtual, en especial cómo Samsung Gear intenta hacerse un hueco y otras compañías tratan de seguirle, y con esto, videojuegos basados en la realidad virtual[4].

Dicho lo anterior, se puede apreciar que los dispositivos móviles, y en especial los smartphones, pueden resultar relativamente interesantes para el desarrollo de videojuegos, no solo para compañías grandes como Nintendo, Ubisoft, o Sega, sino también para desarrolladores independientes, que a pesar de encontrarse en un mercado que ya empieza a estar saturado, y lleno de proyectos con mucha financiación, pueden tener su nicho o un golpe de suerte, como le ocurrió a Dong Nguyen con Flappy Bird, por ejemplo, que llegó a recibir ingresos de 90000 dólares diarios[5]. Por otra parte, es un buen punto de inicio para ganar experiencia y poder publicar los propios desarrollos en grandes plataformas, como Google Play, donde por una licencia de 25 dólares, se puede publicar todas las aplicaciones que se desee durante toda la vida (siempre y cuando cumpla con sus normativas).

Para este Trabajo de Fin de Máster, vamos a desarrollar un videojuego 2D para Android, y utilizaremos para ello el motor de videojuegos Cocos2d-x[6]. Queremos que este proyecto, dadas las características del Máster, sea publicado en Google Play, y haga también uso de sus servicios, como logros, y marcadores. Se desarrollará además el Game Design Document (GDD) del videojuego, para describir todo el diseño del proyecto que vamos a desarrollar.

2 Marco teórico o Estado del arte

En este bloque vamos a hacer un análisis de los motores de videojuegos en su contexto, realizar una comparación entre los motores más populares que existen ahora mismo con aplicación para dispositivos móviles, y también, destacar las utilidades y características que ofrece Cocos2d-x, las cuales han llevado a la elección de esta herramienta para la creación del proyecto que vamos a documentar.

2.1 Concepto de Videojuego

Antes de nada, lo lógico es definir qué es un videojuego, vamos a ello.

Los videojuegos, aunque muchos parecen obras maestras, y probablemente lo sean, al final, su concepto se basa en algo más simple. Como dijo Bernard Suits hace unos años "jugar a un videojuego es un esfuerzo voluntario de superar obstáculos innecesarios"[7]. Por otra parte, Wikipedia nos da la siguiente definición "es un juego electrónico en el que una o más personas interactúan, por medio de un controlador, con un dispositivo dotado de imágenes de vídeo"[8].

Podríamos definir como concepto que un videojuego es una actividad en la que:

- Necesita de al menos un jugador.
- Tiene reglas.
- Tiene una condición de victoria.

Así que probablemente si le damos una definición, sería algo como "un juego que es jugado a través una pantalla de vídeo".

2.2 Motores para producir videojuegos

2.2.1 Motores en su contexto

Antes de comenzar, cabe definir lo siguiente, ¿qué es un motor de videojuegos?

El término motor de videojuegos nace a mediados de los años noventa, en referencia a los primeros juegos en primera persona de disparos (FPS), especialmente el conocido Doom, de id Software. Doom fue diseñado con una arquitectura muy bien definida, contando con una excelente separación entre los componentes del núcleo del motor (como el sistema de renderizado

tridimensional, la detección de colisiones, o el sistema de audio), los recursos artísticos del juego, los mundos de juego, y las reglas para jugarlos.

El valor de esta separación se vuelve evidente cuando los desarrolladores empezaron a crear juegos que partían de esa estructura básica, pero con distinto arte, mundos, armas, vehículos, y otros assets (recursos), además de las reglas de juego, todo esto solo haciendo mínimos cambios al "motor" ya existente. Esto marcó el nacimiento de la comunidad de "mods", un grupo de jugadores individuales y pequeños estudios independientes que construían sus videojuegos modificando juegos ya existentes, usando set de herramientas (toolkits) gratuitos proporcionados por los desarrolladores originales. Y es que gracias a los motores de videojuegos, los diseñadores ya no dependen de los diseñadores, y además es posible añadir o cambiar partes del juego rápidamente, lo cual antes de su invención, podía resultar costoso[9].

A partir del motor de id software, otras empresas decidieron construirse su propio motor de videojuegos, tal como decidió hacer Epic Games con Unreal Engine en 1998[10], o Valve con su motor Source en 2004[11], y otras empresas posteriores.

2.2.2 Comparativa de motores para videojuegos móviles en el mercado

Actualmente existe una lista muy amplia de motores para producir videojuegos, tanto 2D, como 3D, de pago y gratuitos, y que ofrecen mayor o menor compatibilidad con plataformas tanto para desarrollar como para las que producir videojuegos.

Nombrarlos todos sería complicado, dada la cantidad, Wikipedia hace una lista con una gran cantidad de ellos clasificados por licencia:

V · T · E	Game engines (list)	[hide]
Source port · First-person shooter engine (list) · Tile engine · Game engine recreation (list) · Game creation system		
Free software / open source	2D	Adventure Game Studio · Beats of Rage · Box2D · Chipmunk · Cocos2d · Digital Novel Markup Language · Flixel · Exult · Game-Maker · Gosu · Jogre · KiriKiri · Moai SDK · ORX · Pygame · Ren'Py · StepMania · Stratagus · Thousand Parsec · VASSAL · Xconq
	2.5D	Aleph One · Build · Flexible Isometric Free · Id Tech 1 · Wolfenstein 3D
	3D	Away3D · Axiom · Blender Game · Cafe · Crystal Space · Cube · Cube 2 · Delta3D · Dim3 · Genesis3D · GLScene · Horde3D · HPL 1 · Irrlicht · Id Tech 2 · id Tech 3 (Ioquake3) · id Tech 4 · JMonkey · Luxinia · OGRE · Ogre4j · Open Wonderland · Panda3D · Papervision3D · Platinum Arts Sandbox Free 3D Game Maker · PlayCanvas · PLIB · Python-Ogre · Quake · Nebula Device · RealmForge · Retribution · Torque 3D
	Mix	Allegro · Construct Classic · Godot · Lightweight Java Game Library · Spring · Visualization Library
Proprietary	2D	Clickteam Fusion · Coldstone · Construct 2 · Corona · CRX · Fighter Maker · Filmation · GameMaker · GameMaker: Studio · Garry Kitchen's GameMaker · Generic Tile · Gold Box · MADE · Mscape · M.U.G.E.N · NScripter · RPG Maker · Shoot the Bullet · Sim RPG Maker · Sound Novel Tsukūru · Southpaw · Stencyl · Vicious · Virtual Theatre · V-Play · Z-machine · Zillions of Games · ZZT
	2.5D	Genie · INSANE · Infinity · Jedi · Pie in the Sky · Super Scaler · UbiArt Framework
	3D	4A · Advance Guard Game · Anvil/Scimitar · Arsys · Beelzebub · Bork3D · BRender · C4 · Chrome · Creation · CryEngine · Crystal Tools · Dagor · Diesel · Digital Molecular Matter · Disrupt · Dunia · EAGL · EGO · Electron · Elflight · Enforce · Enigma · Essence · Flare3D · Fox · Freescape · Frostbite · Geo-Mod · GoldSrc · HeroEngine · HydroEngine · HPL 2 · id Tech 5 · id Tech 6 · Ignite · Iron · IW · Jade · Kinetica · LS3D · Leadwerks · LithTech · Luminous Studio · LyN · Marmalade · Mizuchi · MT Framework · NanoFX GE · Odyssey · Orochi · Outerra · Panta Rhei · Phoenix Engine (Relic) · Phoenix Engine (Wolfire) · PhyreEngine · Q · Real Virtuality · REDengine · Refractor · RenderWare · Revolution3D · Riot · RAGE · SAGE · Serious · Shark 3D · Silent Storm · Sith · Source · SunBurn XNA · Titan · TOSHI · Truevision3D · Unigine · Unity · Unreal · Vengeance · Visual3D · Voxel Space · XnGine · X-Ray · Yebis · YETI · Zero
	Mix	CPAGE · Dark · Gamebryo · Hybrid Graphics · Kaneva Game Platform · Metismo
Proprietary Game middleware (list)	AiLive · Euphoria · Gameware · GameWorks · Havok · iMUSE · Kynapse · Quazal · SpeedTree · Xaitment	

Figura 1. Lista de motores

Fuente: Wikipedia[12]

Nosotros vamos a destacar y comparar algunos de los más importantes ahora mismo en la producción de videojuegos para dispositivos móviles actualmente. Estos son los más tratados y discutidos durante toda la asignatura de videojuegos en el Máster, Cocos2d-x y Unity.

2.2.3 Cocos2d-x

Figura 2. Logo Cocos2d-x

Fuente: Cocos.org[13]

Cocos2d-x es la versión multiplataforma del motor Cocos2D, diseñado para iOS. Se trata de un motor que funciona con C++, y que permite exportaciones directas a múltiples plataformas, como Android, iOS, Windows Phone, entre otras.

Este motor es Open Source y gratuito, y actualmente lidera su desarrollo y lo mantiene la compañía china Chukong Technologies[14]. Hasta hace poco, Cocos2d-x no era más que unas bibliotecas sin interfaces gráficas que permitieran un desarrollo más amigo, los proyectos se creaban desde un terminal, y las interfaces de usuario, hojas de sprites, y otros, se realizaban por código y usando otras herramientas externas, sin embargo, poco a poco el motor va cogiendo volumen, y ya cuenta con un asistente visual para la creación de proyectos, con una herramienta para crear interfaces, hojas de sprites, y otros, y recientemente acaban de lanzar otra herramienta más para facilitar más la producción gráficamente, llamada Cocos Creator[15].

2.2.4 Unity

Figura 3. Logo Unity

Fuente: Wikimedia[16]

El motor Unity ofrece una amplia gama de características y posee una interfaz sencilla de entender. Además, posee un sistema de integración multiplataforma que permite exportar juegos para casi todas las existentes, siendo actualmente la mejor opción para desarrollo 3D en plataformas de Android, por sus herramientas de compresión que permiten que los videojuegos no sean especialmente pesados, y no consuman excesivos recursos.

El motor de juego es compatible con las principales aplicaciones de modelado y animación 3D, como 3ds Max, Maya, Softimage, Cinema 4D, y Blender, entre otros, lo cual se traduce en que soporta la lectura de archivos exportados con estos programas sin ofrecer ningún problema.

Como contra, Unity es un motor de pago, donde a pesar de desbloquear desde su versión 5.0 casi todas las características que poseía antes solo la versión pro, no deja de ser necesario pagar por royalties si se superan ciertos ingresos.

2.2.5 Elección del motor

Si nos hacemos la pregunta a nivel de desarrollador de qué motor elegir, nos encontramos ante una decisión a considerar, ya que ambos cuentan con una curva de aprendizaje importante y además es necesario familiarizarse con numerosas herramientas. Es por esto que lo recomendable es elegir aquel que se adapte mejor a nuestras necesidades, sin embargo, es conveniente tener en cuenta una serie de detalles[17]:

- Que ofrezca una buena documentación.
- Que tenga una buena comunidad de usuarios que además no se haya abandonado.
- Tener en cuenta si queremos modificar el motor o no.

Para este proyecto hemos optado por utilizar el motor Cocos2d-x que, además de ofrecer el 100% de sus características de forma gratuita, se trata del motor open source para dispositivos móviles más utilizado, con una documentación adecuada, y para un videojuego 2D sencillo de nuestras características es más que suficiente.

2.2.6 Conceptos de la arquitectura de Cocos2d-x

En cocos2d-x cada pantalla se presenta mediante un objeto de tipo Scene. Dentro de ella, se dibujan todos los elementos, como menús, fondos, etc., en forma de nodos de tipo Node, Además, una escena puede estar compuesta de varias capas, que pueden cambiar su posición y orden por la pantalla. Es decir, una escena puede representarse como un grafo con nodos, donde algunos de

ellos como las capas, pueden contener a su vez otros nodos hijos, tal como se puede apreciar en el siguiente grafo:

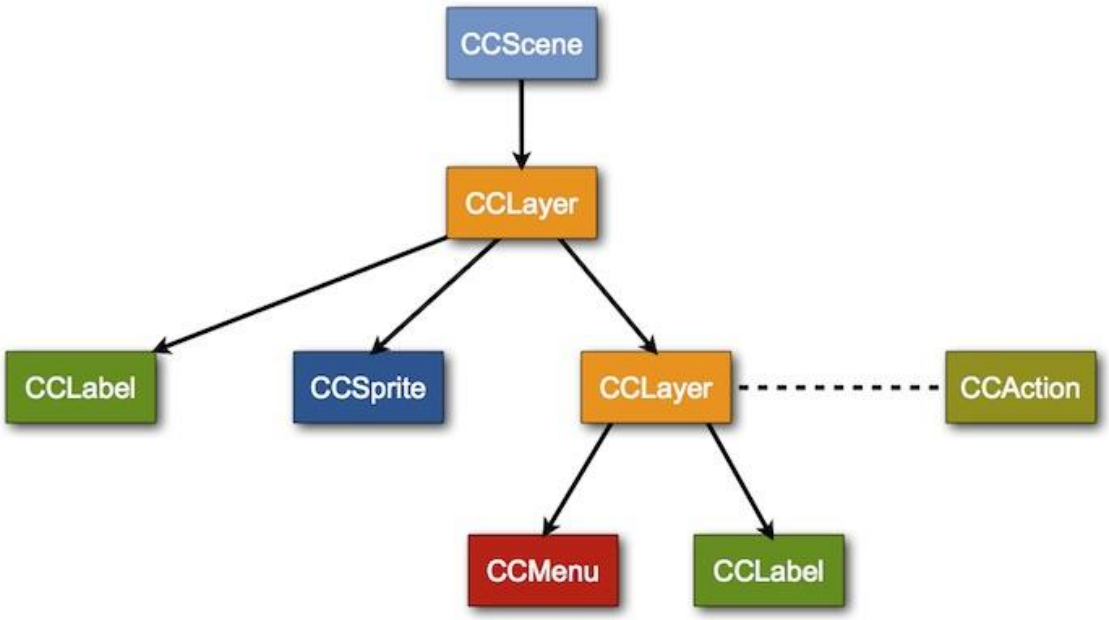


Figura 4. Diagrama Arquitectura Cocos2d-x

Fuente: Libro asignatura Videojuegos Dispositivos Móviles[18]

Con respecto a temas de programación, el origen de Cocos2d-x, como se ha mencionado antes, proviene de la versión original para iOS de Cocos2D, y esto se ve destacado en toda su lógica y funcionamiento, ya que a pesar de programarse en C++, copia el funcionamiento Objective-C, siendo necesario crear objetos de cocos con su función créate, en lugar de como es costumbre en C++, y con métodos de factoría predefinidos a los que se llaman para emular en la mayor medida de lo posible el otro lenguaje.

Por último, es preciso mencionar el elemento central del motor y más importante, un singleton llamado Director, y que nos permitirá principalmente la gestión de escenas.

3 Objetivos

3.1 Objetivo principal del Trabajo de Fin de Máster

lenguaje.

El objetivo principal de este proyecto es la realización de un videojuego 2D sencillo para Android utilizando el motor de videojuegos Cocos2d-x, el cual analizaremos, y comunicar el videojuego con el ámbito social apoyándonos en Google Play Services.

Con este objetivo se pretende realizar un Diseño previo del videojuego, tratar temas sociales, y aprender a utilizar mejor el motor Cocos2d-x.

3.2 Desglose de Objetivos

A continuación se presentan los objetivos del TFM en forma de tareas más específicas:

1. Realizar el GDD (Game Design Document) de un videojuego.
2. Crear varios niveles jugables..
3. Diseño y realización propia de los gráficos del juego.
4. Aprovechar las herramientas de cocos y externas disponibles para llevar a cabo el desarrollo de forma eficaz.
5. Sonorizar el videojuego
6. Comunicar el videojeugo con Google Play Services.
7. Evaluar Cocos2d-x para el desarrollo de videojuegos personal

4 Metodología

4.1 Metodología de desarrollo

La metodología de desarrollo elegida para el desarrollo de este proyecto ha sido Kanban[19], la metodología ágil basada en tableros con etiquetas.

El motivo de esta decisión viene marcado por la escasez de imposiciones sobre el desarrollo que ofrece la metodología, la orientación al número de personas al que va dirigida (normalmente a equipos pequeños), y la facilidad para controlar la carga de trabajo de forma adecuada. Además, como se trata de una metodología ágil, permite un desarrollo menos marcado en fases específicas y que soporta cambios con mayor facilidad que otro tipo de metodologías más cerradas o con unas especificaciones de diseño demasiado estrictas.

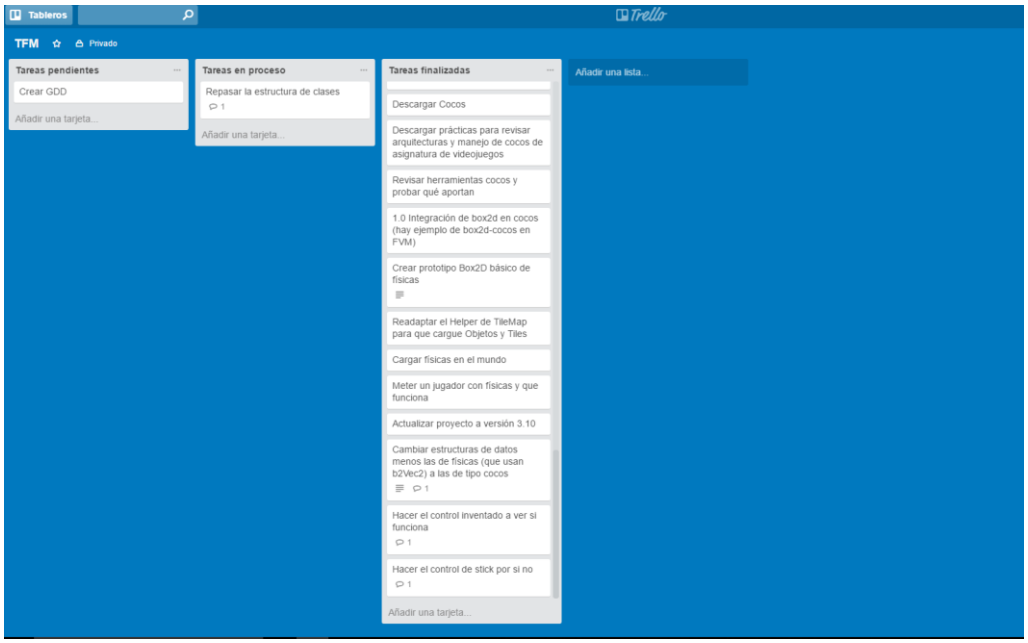


Figura 5. Captura de la organización Kanban del proyecto

Fuente: Elaboración propia.

El flujo de trabajo en este proyecto ha consistido en añadir nuevas tareas pendientes al tablero. Cuando se pretendía realizar alguna o varias tareas, se colocaban en una lista de tareas en proceso, y, cuando acababan, se colocaban por último en una lista de tareas terminadas.

4.2 Gestión del proyecto

Para la gestión de tareas del proyecto, se ha decidido utilizar la aplicación Trello, que permite la creación de un tablero y la organización de tareas por etiquetas. Las tareas en esta aplicación soportan una variedad de opciones que resultan cómodas en el momento de definir tareas o añadir modificaciones o notas a estas (en forma de comentarios en nuestro caso particular, al solo ser una persona). Como es multiplataforma, permite la gestión rápida de estas desde smartphones o tablets, para hacer posibles cambios o añadir nuevas tareas en cualquier momento.

4.3 Control de versiones y repositorio

Dado que el proyecto solamente tiene un desarrollador, se ha optado por trabajar sin hacer uso de ningún repositorio a la hora de realizar el proyecto. En momentos clave del desarrollo, como la creación del primer mundo funcional, o la inclusión de redes sociales, se realizaron copias de seguridad locales para evitar posibles riesgos.

5 Cuerpo del trabajo

El cuerpo del trabajo de desarrollo queda dividido en dos bloques principales.

1. Un primer bloque dedicado al **Documento de Diseño del Videojuego (GDD)**, donde se cubren todos los aspectos del diseño del videojuego desarrollado.
2. Y un segundo bloque, de **Desarrollo e implementación**, correspondiente a los aspectos técnicos más relevantes de la producción del proyecto de estudio.

5.1 Documento de Diseño del Videojuego (GDD)

Figura 6. Imagen portada del videojuego.

Fuente: Elaboración propia

En este bloque se describe con detalle todos los aspectos de diseño necesarios para la correcta implementación posterior del videojuego desarrollado. Se trata tanto historia, experiencia de juego, y diseño de niveles, como interfaz de usuario y detalles técnicos.

Este documento está basado de acuerdo a la plantilla de GDD ofrecida en la asignatura de Videojuegos para Dispositivos Móviles, del segundo cuatrimestre en el Máster.

El nombre que se ha decidido dar al proyecto es **Elegante Johns y Pájaro Peligro**.

5.1.1 El juego en términos generales

5.1.1.1 Resumen del argumento

Elegante Jonhs es un tipo elegante, siempre trajeado, ese tipo de personas que no puede faltar en una convención de gente elegante, en reuniones de negocios, citas formales, y otras actividades en las que se prime la elegancia. Sin embargo, hay un pájaro muy travieso que no quiere que Elegante Johns vaya a sus citas elegantes, este es Pájaro Peligro, que intentará hacer caca sobre Elegante Johns y estropear sus planes, para que así se convierta en Sucio Johns.

Pero Elegante Johns no se rendirá, su objetivo no es otro que volverse uno de los grandes elegantes de su país, Elegantelandia, y para ello necesita conseguir los 5 paraguas elegantes, situados en 5 zonas del mundo.

Cuando elegante Johns consiga los 5 paraguas, recibirá el título de Lord Elegante, o si demuestra elegancia absoluta, podrá acceder a una zona que contiene un paraguas secreto que le permitirá ser un King Elegante, un título de elegancia de ensueño.

5.1.1.2 Conjunto de características

Las principales características atractivas del videojuego son las siguientes:

- **Atractivo** apartado visual con estilo de dibujos sobre papel.
- Sencillo pero adictivo.
- Desarrollado con el motor open source **Cocos2d-x**.

5.1.1.3 Género

Elegante Johns y Pájaro Peligro se trata de un juego en tercera persona, de plataformas por niveles repartidos en zonas, y de tipo side-scrolling, donde el personaje se desplaza automáticamente hacia delante. Un ejemplo de juego de este estilo son los juegos de Rayman para Android. Se pretende que sea un juego casual, es decir que no requiera de mucho tiempo de aprendizaje ni dedicación para dominarlo.

5.1.1.4 Audiencia

Dadas las características del diseño, el juego se encuentra dentro de una clasificación por edades de tipo PEGI3, de acuerdo a la descripción ofrecida por la web oficial de PEGI sobre PEGI3:

Figura 7. Logo PEGI3.

Fuente: Wikimedia[20]

> *"El contenido de los juegos con esta clasificación se considera apto para todos los grupos de edades. Se acepta cierto grado de violencia dentro de un contexto cómico (por lo general, formas de violencia típicas de dibujos animados como Bugs Bunny o Tom y Jerry). El niño no debería poder relacionar los personajes de la pantalla con personajes de la vida real, los personajes del juego deben formar parte exclusivamente del ámbito de la fantasía. El juego no debe contener sonidos ni imágenes que puedan asustar o amedrentar a los niños pequeños. No debe oírse lenguaje soez."[21]*

A pesar de esto, el videojuego es apto para personas de cualquier edad posterior.

5.1.1.5 Resumen del flujo de juego

Elegante Johns y Pájaro Peligro es un juego 2D de tipo side-scrolling basado en niveles. En este juego encarnamos a Elegante Johns, con el que debemos superar cada una de las misiones de elegancia en cada una de las zonas del país (Elegantelandia) con el fin de obtener los paraguas elegantes. Conforme se supera niveles, se desbloquean nuevos hasta superarlos todos.

Dentro de cada nivel, el jugador puede lograr mayor o menor elegancia, esto es, recogiendo mientras supera el nivel todas las monedas elegantes que encuentre.

5.1.1.6 Apariencia del juego

En cuanto a la apariencia (experiencia) del juego, se pretende destacar un aspecto burlón que resulte relativamente gracioso.

Por un lado, el apartado gráfico está basado en dibujos a mano sobre papel, lo que da un aspecto más rudimentario y burlesco, y junto con el tronco argumental de la historia, le otorgan un nivel de absurdez que llama a cierto humor.

Por otra parte, la banda sonora busca potenciar también la esencia cómica del videojuego.

5.1.1.7 Ámbito

El videojuego se desarrolla en un mundo imaginario, con 6 niveles gráficamente muy distintos entre sí, pero que, a su vez, comparten el mismo sistema de juego. El juego no contiene inteligencia artificial destacable, ya que se basa en lograr llegar al final de cada nivel evitando todos los obstáculos que nos lo impidan.

5.1.2 Jugabilidad y mecánicas

5.1.2.1 Jugabilidad

Antes de adentrarnos en temas de jugabilidad, es conveniente saber lo que es, una clara definición es la que proporciona la RAE:

> *"Facilidad de uso que un juego, especialmente un videojuego, ofrece a sus usuarios."[22]*

Que básicamente quiere decir que cuanto más cómodo sea jugar a un videojuego, más jugable es.

5.1.2.1.1 Objetivos del juego

En Elegante Johns y Pájaro Peligro el objetivo principal del juego es superar todos los niveles llegando hasta el final de cada uno de ellos.

Por otra parte, como objetivos secundarios para realizar, el jugador puede completar los logros establecidos en el apartado Logros.

5.1.2.1.2 Progresión

La progresión en el juego viene marcada por distintos factores:

- Desbloquear nuevos niveles.
- Aumentar muertes y monedas por cada nivel superado, y también repetido (si vuelve a jugar un nivel, se le suman las monedas y muertes a las que ya tiene).
- La obtención de logros.

5.1.2.1.3 Controles de juego

Como buscamos la mayor sencillez posible en el control, se ha optado por minimizarlos al máximo, al fin y al cabo, un chaval con 3 años tiene que ser capaz de jugar. Esto es:

Figura 8. Control

Fuente: Elaboración propia.

- Para todos los menús: navegación táctil mediante toques.
- Para los niveles:
    - El personaje se mueve automáticamente hacia la derecha.
    - La pantalla se divide en dos polos o sectores, izquierdo y derecho. Si el jugador toca el lado derecho de la pantalla, el personaje salta, y si toca el izquierdo, se mueve más rápido, siendo posible combinar ambos.

Además, si el jugador pulsa el botón back de su terminal, volverá hacia atrás entre los menús, y si está en un nivel, se activará la pausa.

5.1.2.2 Mécanicas

Planteamos tres mecánicas básicas para este juego:

1. Monedas (Collectables).

Figura 9. Gráfico moneda
Fuente: Elaboración propia

Son elementos que pueden recogerse en los niveles, y que el jugador acumula.

2. Cacas (Hazards).

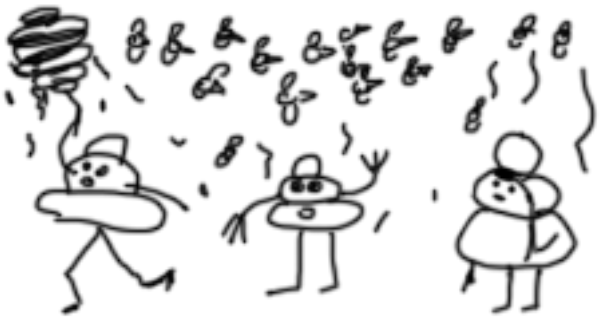


Figura 10. Uno de los muchos gráficos de Cacas.
Fuente: Elaboración propia.

Los niveles están llenos de cacas con patas y ojos en distintas situaciones, si el jugador toca una, muere.

3. Paraguas (Fin de nivel).

Figura 11. Gráfico Paraguas.
Fuente: Elaboración propia.

Cuando el jugador llegue hasta él, supera el nivel en el que se encuentre.

5.1.2.3 Rejugar y salvar

En cuanto a la rejugabilidad y el guardado de partida, vayamos por partes.

Como se ha hecho mención anteriormente, es posible rejugar los niveles cuantas veces el jugador quiera, y cada vez que estos se superen, suman puntos al jugador, es decir, constantemente puede ir sumando monedas y muertes, y esto, como es lógico, lo vamos a aprovechar con los logros para darle un toque extra de rejugabilidad a Elegante Johns y Pájaro Peligro.

Dicho esto, tanto esta acumulación como el desbloqueo de niveles conlleva cierta persistencia, y como buscamos la menor complicación para el jugador, el juego presentará un guardado automático de muertes y monedas conseguidas, y también de los niveles desbloqueados.

5.1.2.4 Logros

Como logros iniciales, se ha decidido plantear los siguientes, que tratan de incentivar lo máximo posible que el jugador siga jugando:

1. Elegantín. Superar el nivel 1.
2. Algo Elegante. Superar el nivel 2.
3. Elegante. Superar el nivel 3.
4. Considerablemente elegante. Superar el nivel 4.
5. Muy Elegante. Superar el nivel 5.

6. King Elegante. Superar el nivel 6.
7. Para chuches. Conseguir al menos 10 monedas.
8. Entradas + Menú en el cine. Conseguir al menos 25 monedas.
9. A mí no me digas el menú, dame la carta. Conseguir al menos 50 monedas.
10. El Humilde. Conseguir al menos 100 monedas.
11. Una nueva amiga. Morir al menos 10 veces.
12. La pienso mucho, creo que me enamoré de ella. Morir al menos 50 veces.
13. Nacido para morir. Morir al menos 100 veces.

Para poder desbloquearlos es necesario que el jugador se conecte con la plataforma de servicios que le solicitemos.

5.1.3 Historia, características, y personajes

5.1.3.1 Historia

Corría el año 2020 cuando la sociedad llegó hasta puntos inhóspitos de ordinariez y salvajismo, todos, zombis en sus distintas burbujas, dejaron de tener cada vez menos respeto y vergüenza en el mundo en favor de algo de atención en un mundo completamente inmerso en las redes.

Ante estos problemas, invisibles para todos, solo algunos pudieron percatarse del fin inminente, algunas personas que seguían creyendo en los valores de la elegancia, para los que llevar traje y corbata significaba todavía algo, un grupo de personas suficientemente grande y que pudieron recaudar capital suficiente como para comprar y fundar su propio país, Elegantelandia, el país elegante, con sus propias leyes, donde la primera de todas era la obligación de los ciudadanos la de mantener la elegancia.

Un año después de la creación de Elegantelandia, con todas las personas elegantes en el mismo país, ya no quedaba gente en el resto del mundo que supiera hacer cosas, así que colapsó por completo, y todas las personas se convirtieron en caca con patas y ojos, a excepción de alguien, Pájaro Peligro, que ya que era distinto y más guay, decidió apoderarse del mundo.

Pájaro Peligro no tuvo gran dificultad en dominar el mundo, y ser capaz de transportar caca a todas las partes que quisiera, así que puso su empeño en conquistar Elegantelandia, el último bastión humano en un mundo de caca.

Para llevar a cabo tal acción, Pájaro Peligro se propuso llenar Elegantelandia de cacas tanto como pudiera, sin embargo, todavía había gente que se resistía, gente como Elegante Johns, que aspira a seguir con su estilo de vida, y aprovechar la situación para demostrar su elegancia ante las adversidades.

5.1.3.2 Mundo de juego

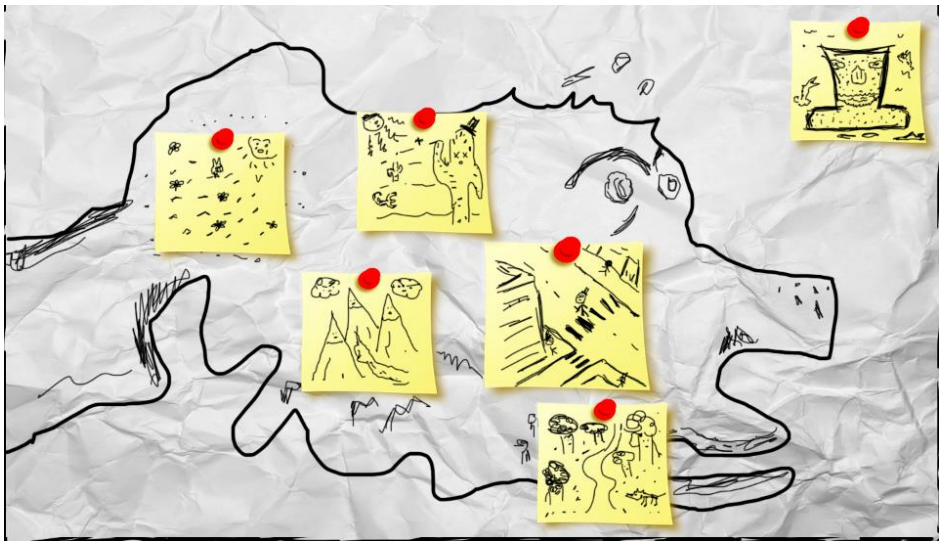


Figura 12. Gráfico del mapa de Elegantelandia.

Fuente: Elaboración propia.

La historia transcurre en el país de Elegantelandia, una península con todo tipo de ambientes distintos, y una gran ciudad donde convive toda la gente elegante.

5.1.3.3 Personaje principal. Elegante Johns

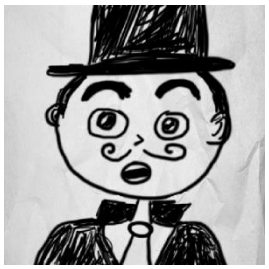


Figura 13. Gráfico del personaje de Elegante Johns.

Fuente: Elaboración propia.

Elegante Johns es un hombre elegante de 22 años de edad, recién ha terminado sus estudios en la universidad de la elegancia y quiere formar parte de la élite elegante, su sueño desde que era un niño.

Le gustan los paseos bajo la lluvia y tomar leche con cacao por la noche antes de dormir, le relaja y además mantiene el estómago ocupado por la noche para no despertarse a las 3:00 am con hambre.

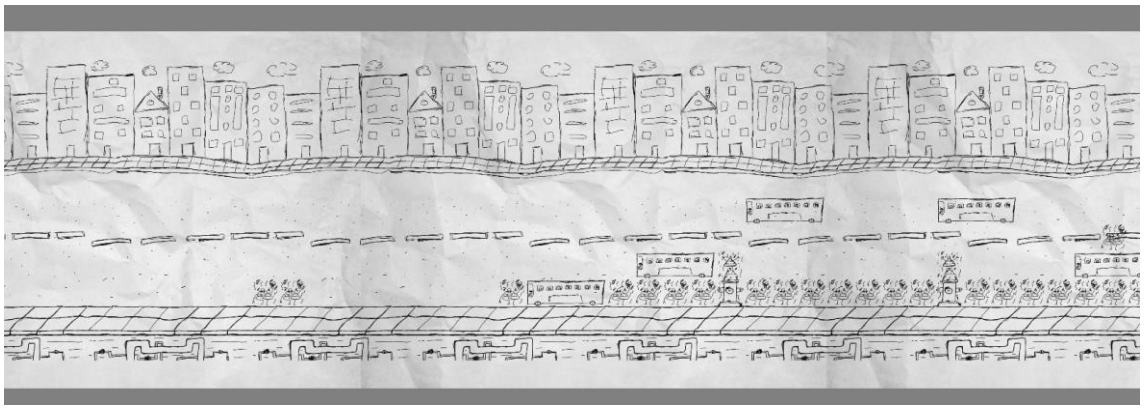
A veces le cuesta mantener la compostura, pero es un tipo valiente y con mucha energía.

5.1.4 Niveles de juego

Los niveles del videojuego comparten todos la misma jugabilidad y mecánicas, cambiando gráficos y disposición de elementos.

Como se ha explicado anteriormente, el jugador comenzará en una punta y deberá atravesar todo el escenario, hasta llegar al final, donde se encontrará con un paraguas que corresponde al fin del nivel.

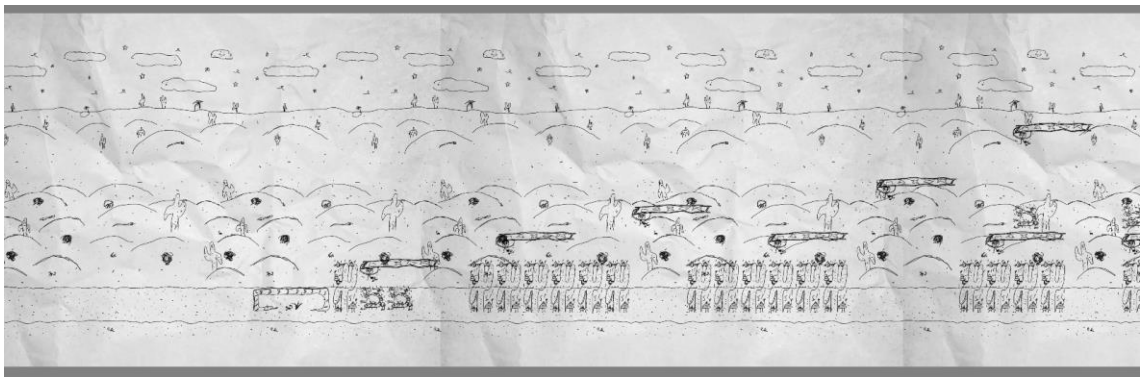
5.1.4.1 Zona 1. Ciudad



Figrua 14. Sección de un trozo del mapa de la Ciudad.

Fuente: Elaboración propia.

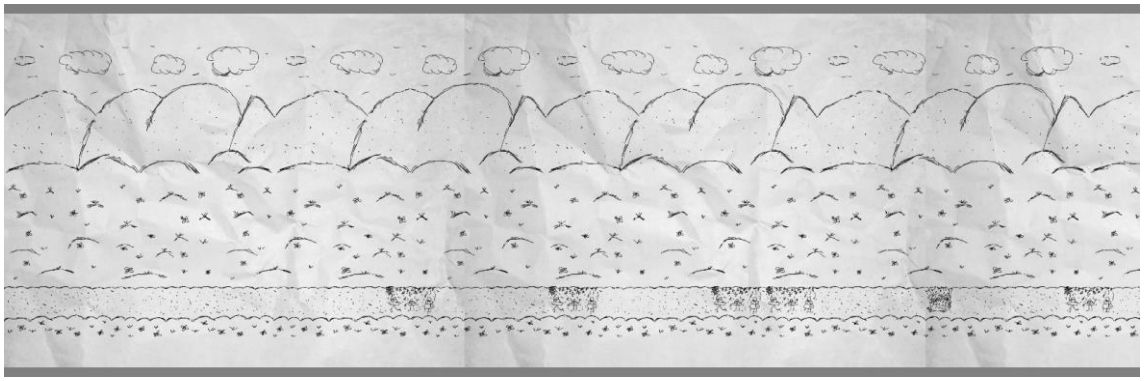
5.1.4.2 Zona 2. Desierto



Figrua 15. Sección de un trozo del mapa del Desierto.

Fuente: Elaboración propia.

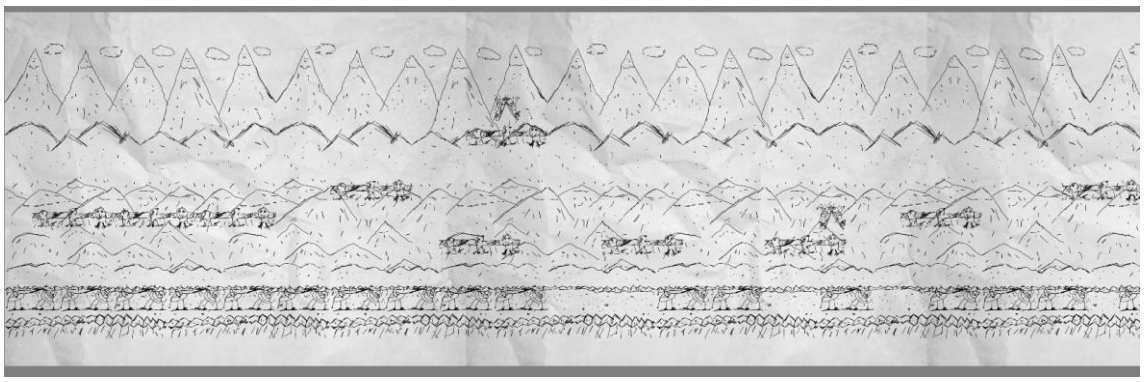
5.1.4.3 Zona 3. Pradera



Figrua 16. Sección de un trozo del mapa de la Pradera.

Fuente: Elaboración propia.

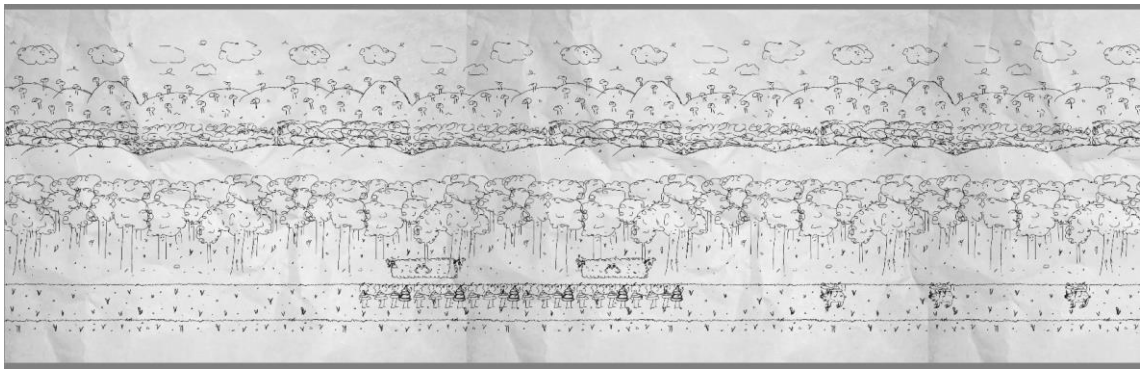
5.1.4.4 Zona 4. Montañas



Figrua 17. Sección de un trozo del mapa de las Montañas.

Fuente: Elaboración propia.

5.1.4.5 Zona 5. Bosque



Figrua 18. Sección de un trozo del mapa del Bosque.

Fuente: Elaboración propia.

5.1.4.6 Zona 6. Isla Elegante

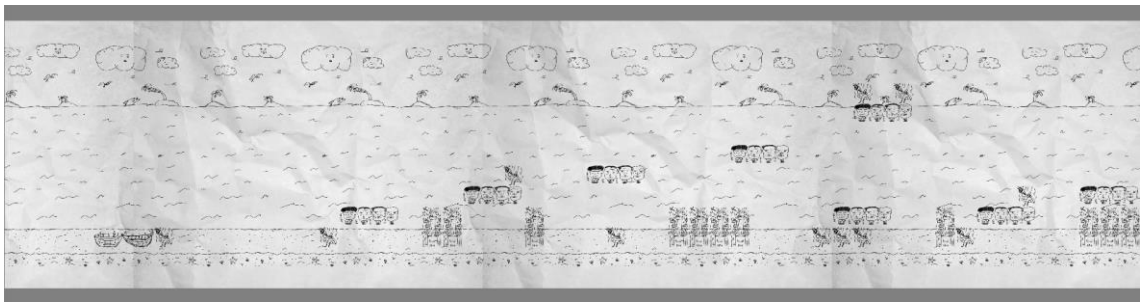


Figura 19. Sección de un trozo del mapa de la Isla Elegante.

Fuente: Elaboración propia.

5.1.5 Interfaz

5.1.5.1 HUD

El HUD (heads-up display), es información 2D que aparece dibujada sobre la pantalla de juego, mostrando información útil al jugador.

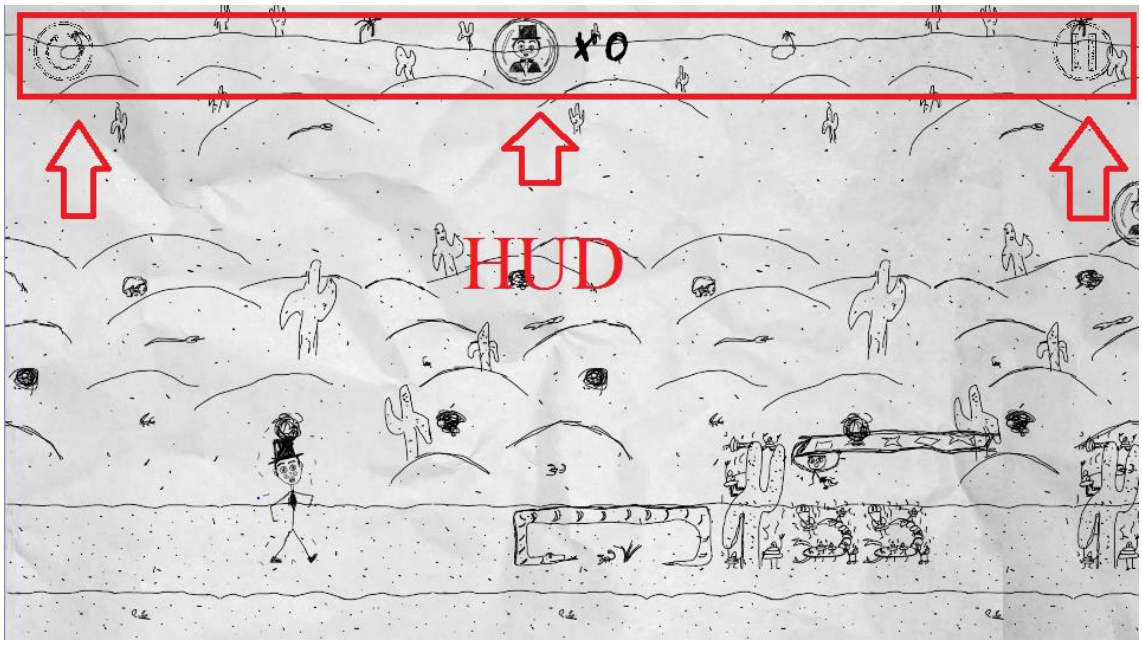


Figura 20. Captura Ingame del juego con retoque para indicar el HUD

Fuente: Elaboración propia.

En Elegante Johns y Pájaro Peligro, el HUD muestra la cantidad de monedas obtenidas durante la partida, en la parte superior de la pantalla, junto con los botones de reiniciar nivel y de pausar la partida, en cada extremo.

5.1.5.2 Menús

Los menús nos permitirán acceder al nivel de juego, modificar aspectos del juego (resolución y volumen general), acceder a niveles de extras, salir del juego, pausarlo, o acceder a los créditos, entre otras cosas.

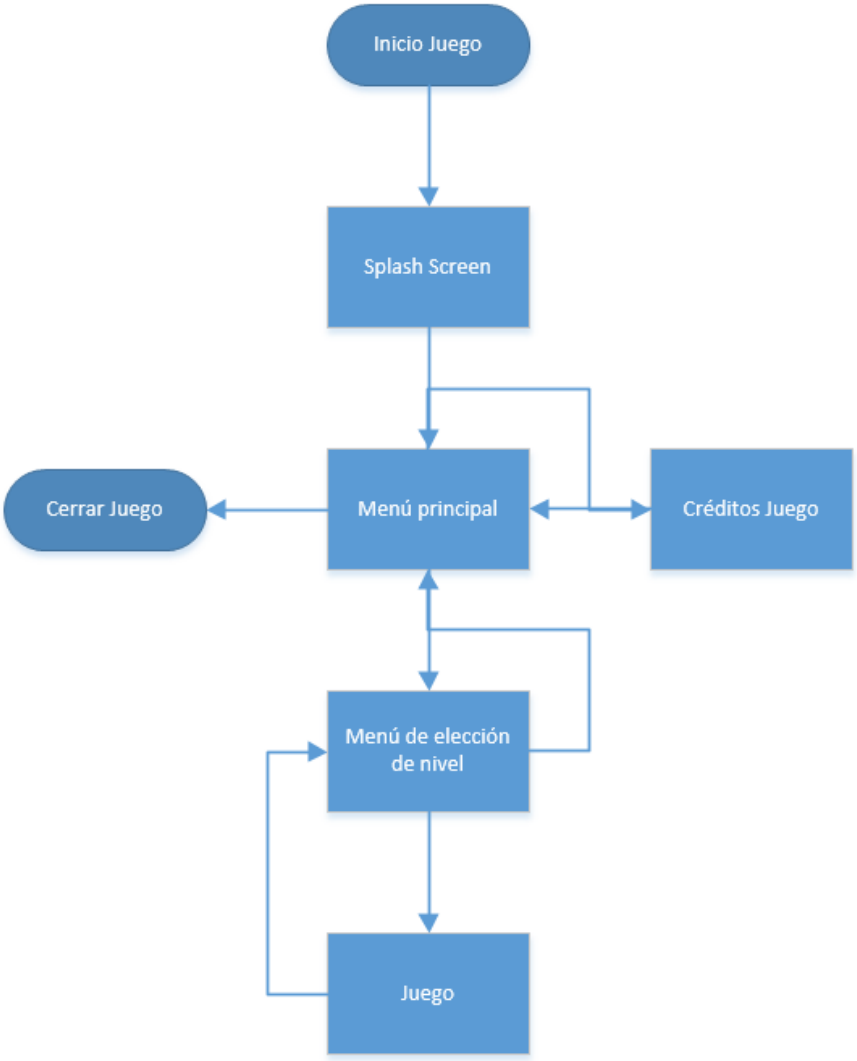


Figura 21. Diagrama básico del flujo de juego.

Fuente: Elaboración propia.

A continuación vamos a explicar los menús utilizados en este videojuego.

5.1.5.2.1 Splash Screen

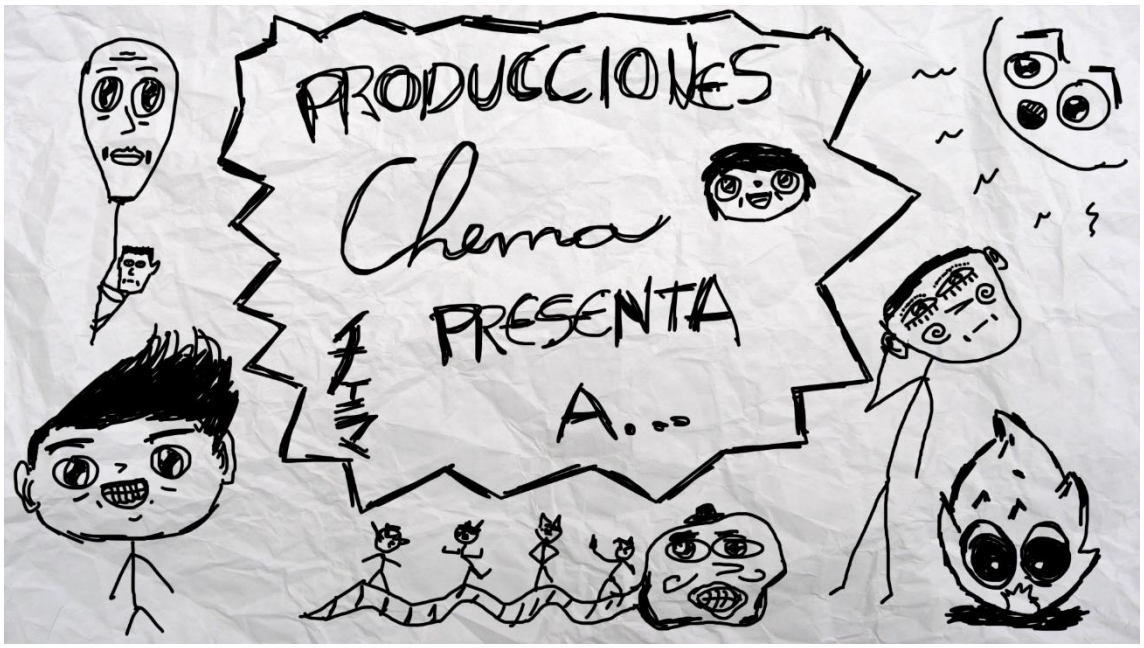


Figura 22. Captura de la pantalla Splash.

Fuente: Elaboración propia.

Este menú es más bien una pantalla temporal que aparece al iniciar la aplicación, en ella se muestra información sobre el creador, la tecnología de desarrollo, y en nuestro caso aprovecharemos para precargar algunos gráficos.

5.1.5.2.2 Menú principal

Figura 23. Captura del Menú principal.

Fuente: Elaboración propia.

Dentro del menú principal, que como su nombre indica, es el esencial, podremos acceder al menú de selección de nivel, a los créditos de juego, y también consultar los logros y marcadores si nos hemos conectado con el servicio externo. Además, desde este menú es desde donde podemos salir de la partida.

5.1.5.2.3 Menú de créditos

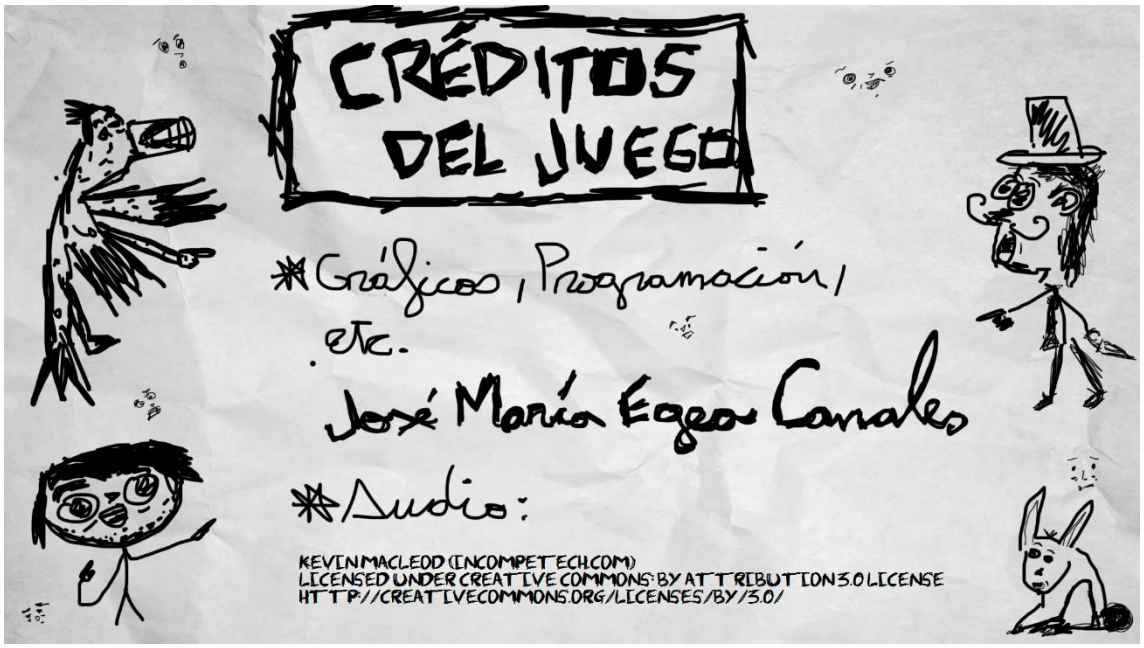


Figura 24. Captura del menú de créditos.

Fuente: Elaboración propia.

Utilizamos este menú para mostrar información sobre quién es el creador del juego, e indicar si hemos utilizado algún componente de un tercero, bien sea algún audio o alguna música que necesite ser referenciada.

5.1.5.2.4 Menú de selección de nivel

Figura 25. Captura del menú de selección de nivel.

Fuente: Elaboración propia.

Desde este menú es donde el jugador elige qué nivel de juego quiere cargar.

5.1.5.2.5 Submenús ingame y otros

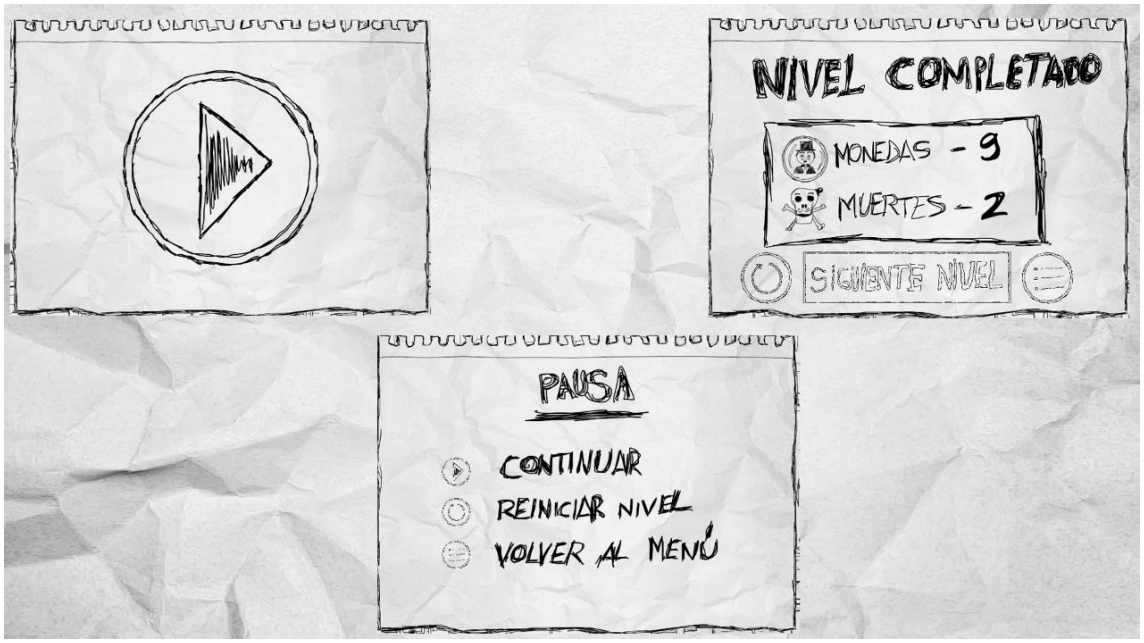


Figura 26. Submenús de Comenzar, Pausa, y Nivel Completado.

Fuente: Elaboración propia.

Dentro del juego encontramos algunos submenús, tales como el submenú de pausa, que nos permite, como su nombre indica, pausar la partida (y también reanudarla o salir), también los submenús de comenzar la partida (el nivel no comienza hasta que no se pulsa este botón), y el submenú de fin del nivel, que permite acceder al siguiente nivel, repetir el actual, o acceder al menú de selección. Por otro lado, también contamos con submenús automáticos generados por el servicio de logros externo, y cuyo diseño depende de este en nuestro caso.

5.1.5.3 Cámara

En el juego contamos con una cámara ingame. Esta cámara de juego permanece estática en el eje Y, y se encarga de seguir al personaje a lo largo del eje X en todo el nivel.

5.1.5.4 Sonido

En Elegante Johns básicamente todo cuenta con su acompañante sonoro, tanto pulsar botones, como las acciones del personaje, y tanto los menús como en cada nivel, contamos con música que acompañe la situación.

Dado el tono relativamente humorístico del juego, las canciones y sonidos buscan potenciar este efecto para proporcionar la mejor experiencia posible.

Por comodidad, dividimos la banda sonora en mfx para las canciones, y sfx para los efectos de sonido.

5.1.6 Guía Técnica

5.1.6.1 Requerimientos de Hardware

Según las especificaciones de cocos2d-x, para poder ejecutar los videojuegos en un terminal Android, es necesario contar con un dispositivo con una versión 2.3 o superior.

Para el desarrollo, como se ha realizado utilizando un terminal Windows, los requisitos mínimos que se exigen por el momento son tener un sistema operativo con Windows 7 o superior, y tener instalado Visual Studio 2012 o superior.

5.1.6.2 Software

En cuando al software que se pretende utilizar para el desarrollo del proyecto:

- Para la programación utilizamos Visual Studio.
- Para la creación y edición de gráficos utilizamos Photoshop.
- Para la creación de los spritesheets utilizamos Cocos Studio.
- Para la creación de los niveles de juego utilizamos Tiled.
- Para cualquier edición de audio utilizamos Audacity.

5.2 Desarrollo e implementación

5.2.1 Creación del proyecto.

Para la creación de un proyecto con el que poder trabajar solo es necesario descargarse el motor desde la página oficial, y este al abrirlo muestra una lista con los últimos proyectos creados, y tiene un asistente que permite crear nuevos, con solo seguirlo.

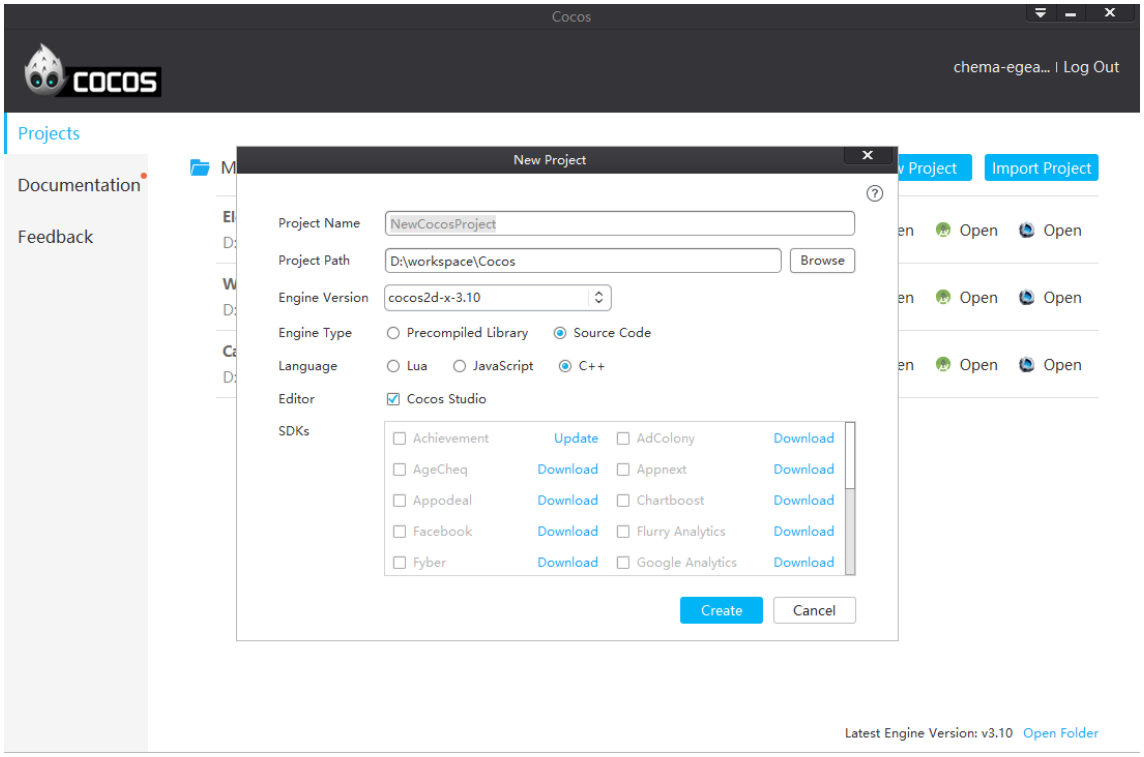


Figura 27. Captura de la creación de un proyecto nuevo en Cocos2d-x usando el asistente.

Fuente: Elaboración propia.

En nuestro caso, hemos creado un proyecto en C++, y con nada precompilado, porque para Windows esta opción no funcionaba al menos en la versión de cocos utilizada (3.10).

Una vez creado el proyecto, solo es necesario ejecutarlo y ya viene todo configurado para poder empezar a trabajar.

5.2.2 Engine personalizado

Se ha utilizado una estructura base personalizada para la lógica de los niveles. Esta base está tomada de la plantilla base que se proporciona en la asignatura de Videojuegos para Dispositivos Móviles del máster.

Esta plantilla nos ayuda a gestionar las entidades de la escena, y para eso contamos con una clase que hereda de Scene con algunos métodos esenciales para cada escena que creemos, un par de clases entidad con nodo propio y métodos básicos que nos facilitan su gestión, y por último, una clase Engine2d que se encarga de organizar adecuadamente en una clase la configuración que cocos hace en bruto en la clase AppDelegate (clase donde empezamos a tener el control sobre lo que ocurre en el juego).

5.2.3 Implementación de los menús.

```
73      //BOTON LOGROS
74      auto bLogros = MenuItemImage::create(IMG_B_LOGROS, IMG_B_LOGROS_PRESIONADO, IMG_B_LOGROS_BLOQUEADO, CC_CALLBACK_1(EJ_MenuPrincipal::LoadLogros, this));
75      bLogros->setAnchorPoint(Vec2(0.5, 0.5));
76      bLogros->setPosition(1147, 160);
77      bLogros->setScale(0.15f);
78
79      //BOTON SALIR
80      auto bSalirJuego = MenuItemImage::create(IMG_B_SALIRJUEGO, IMG_B_SALIRJUEGO_PRESIONADO, IMG_B_SALIRJUEGO_BLOQUEADO, CC_CALLBACK_1(EJ_MenuPrincipal::ExitGame, this));
81      bSalirJuego->setAnchorPoint(Vec2(0.5, 0.5));
82      bSalirJuego->setPosition(100, 995);
83      bSalirJuego->setScale(0.14f);
84
85      //BOTON GOOGLE
86      auto bGoogle = MenuItemImage::create(IMG_B_GOOGLE, IMG_B_GOOGLE_PRESIONADO, IMG_B_GOOGLE_BLOQUEADO, CC_CALLBACK_1(EJ_MenuPrincipal::ToogleGooglePlayServices, this));
87      bGoogle->setAnchorPoint(Vec2(0.5, 0.5));
88      bGoogle->setPosition(1820, 985);
89      bGoogle->setScale(0.13f);
90
91      //CREAR EL MENU CON LOS BOTONES
92      auto menu = Menu::create(bJugar, NULL);
93      menu->addChild(bCreditos);
94      menu->addChild(bSalirJuego);
95      menu->addChild(bGoogle);
96      menu->addChild(bMarcadores);
97      menu->addChild(bLogros);
98      menu->setPosition(Point::ZERO);
99
100     this->addChild(menu);
101
```

Figura 28. Captura de la creación de parte del menú principal.

Fuente: Elaboración propia

La creación de los menús se ha realizado mediante programación para tener un mayor control y mantener un código más limpio. Para crear los menús se parte de la clase de cocos MenuItem, que cuenta con varios subtipos de los que podemos hacer uso. Para este proyecto se ha utilizado MenuItemImage como botones, y MenuItemLabel para campos de texto, y MenuItemSprite para fondos de menús.

El funcionamiento es sencillo, tanto MenuItemImage como MenuItemSprite reciben 3 imágenes, una para su estado normal, una para su estado bloqueado, y otra para su estado seleccionado, la diferencia está en que con MenuItemImage contamos con un último parámetro, que es un callback para indicar qué queremos que ocurra cuando se pulse sobre el elemento. En cuanto a MenuItemLabel, funciona como un elemento que tiene un label de texto, y este lo podemos modificar cambiando su texto, su color, su tamaño, etc.

Una vez se crean los elementos del menú, se procede a crear un menú que los contenga, y finalmente se añade a la escena con addChild.

5.2.4 Implementación del gameplay.

Para la implementación del gameplay (la jugabilidad), que en este caso nos referimos a la forma de jugar los niveles, se ha optado por seguir una estructura basada en dos clases, una clase Game, que se trata de una escena que hereda del motor personalizado indicado anteriormente, y que contendrá los distintos submenús (menú pausa, comenzar, fin nivel), el personaje, el control de juego, y un objeto de la otra clase importante, una clase mundo, que también hereda del motor personalizado, y contiene el nivel de juego cargado.

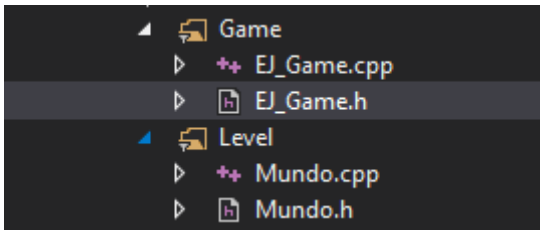


Figura 29. Captura de las dos clases del proyecto más relevantes a la hora de cargar un nivel.

Fuente: Elaboración propia.

El funcionamiento es sencillo, la clase Game es la Scene, y de ella parten todos los nodos hijos mencionados, que a su vez, como en el caso de Mundo, contiene subhijos, correspondientes a las monedas, los peligros, el fin de nivel, las físicas, etc.

```
void preloadResources();
void start();
void resetLevel();

void updateEachFrame(float delta);
void updateIA(float delta);

void updateMonedas(Collectable *m);
Node* getScrollable();

// Controles virtuales (pantalla)
void onButtonPressed(Button button);
void onButtonReleased(Button button);

//Controles fisicos (atras del movil)
void onKeyPressed(EventKeyboard::KeyCode keyCode, cocos2d::

//Menus
Menu * m_menuIngame;
Menu * m_menuPausa;
Menu * m_menuComenzarPartida;
Menu * m_menuNivelSuperado;

//Creacion de menus
void crearMenuIngame();
void crearMenuPausa();
void crearMenuComenzarPartida();
void crearMenuNivelSuperado();

//Lo primero que se carga cuando empezamos el nivel
void prepareToAbrirMenuComenzarPartida();
void abrirMenuComenzarPartida();
void ocultarMenuIngame();
//Actions de los botones
void PausarNivel(cocos2d::Ref *sender);
void ReiniciarNivel(Ref *sender);
void ContinuarPartida(Ref *sender);
void ComenzarPartida(Ref *sender);
void AbrirMenuFinNivel(Ref *sender);
void SiguienteNivel(Ref *sender);
void VolverMenuSeleccionNivel(Ref *sender);
```

Figura 30. Captura de la parte pública de la clase Game (se aprecian los menús).

Fuente: Elaboración propia.

Dicho lo anterior, Game contiene el control, que es una clase aparte que hemos creado para una mejor gestión, y que hace saltar eventos con los parámetros que hemos definido en el GDD (una mitad de pantalla es un botón, y la otra, otro). Con esto, nos dedicamos a indicar qué queremos que ocurra en función de estos eventos cuando salten, y son cambios en el personaje, en concreto, que se ponga a correr o no, y que salte.

```
EJ_Player * m_player;
Mundo *m_mundo;
VirtualControls *m_input;
EJ_AudioHelper m_audio;
```

Figura 31. Un sector de la parte privada de la clase Game (apreciar que contiene a todo).

Fuente: Elaboración propia.

5.2.5 Personaje

El personaje es el que se va a encargar de interpretar todas las colisiones que tenga, esto se hace mediante sensores que controlen si se produce alguna colisión.

```
// ********************************
//  Sensor de tocar suelo
// ********************************

b2PolygonShape shapeSensorDown;
shapeSensorDown.SetAsBox(widthptm * 0.25, heightptm * 0.05, b2Vec2(-widthptm * 0.1, heightptm* (0.1 - 0.55)), 0);

m_groundTest = m_body->CreateFixture(&shapeSensorDown, 1.0);
m_groundTest->SetSensor(false);
m_groundTest->SetUserData(m_userDataCollisionsPlayer);

// ********************************
//  Sensor de colision con algo a la derecha
// ********************************

b2PolygonShape shapeCollisionSensor;

//Creamos la fixture
shapeCollisionSensor.SetAsBox(m_playerSprite->getContentSize().width * 0.1 / PTM_RATIO, m_playerSprite->getContentSize().height * 0.3 / PTM_RATIO, b2Vec2(m_playerSprite->getContentSize

m_frontalTest = m_body->CreateFixture(&shapeCollisionSensor, 1.0);
m_frontalTest->SetSensor(true);

m_frontalTest->SetUserData(m_userDataCollisionsPlayer);

// ********************************
//Indicamos quien es la clase contact listener
// ********************************

world->SetContactListener(new PlayerContactListener(this));
```

Figura 32. Captura de la clase jugador de la parte donde se crean y añaden los sensores.

Fuente: Elaboración propia.

Aparte de esto, desde el personaje, en función del control, como se ha comentado antes, se hacen cambios entre las distintas acciones que este tiene (moverse, correr, morir, saltar, estar quieto).

```
56      //Controles virtuales (pantalla)
57      void EJ_Game::onButtonPressed(Button button)
58      {
59          if (button == Button::BUTTON_ACTION)
60          {
61              m_player->Jump();
62          }
63          if (button == Button::BUTTON_RUN)
64          {
65              m_player->Run();
66          }
67      }
68      void EJ_Game::onButtonReleased(Button button)
69      {
70          if (button == Button::BUTTON_RUN)
71          {
72              m_player->EndRun();
73
74          }
75      }
```

Figura 33. Captura de la clase Game que indica al personaje qué hacer cuando se pulse el control.

Fuente: Elaboración propia.

Por último, el personaje es un ser animado, y es en su clase donde le cargamos todas sus animaciones y las reproducimos en función de la acción que esté realizando.

```
//Animacion Andar
animWalk = Animation::create();
animWalk->addSpriteFrame(spriteFrameCache->getSpriteFrameByName("EJ_Walking1.png"));
animWalk->addSpriteFrame(spriteFrameCache->getSpriteFrameByName("EJ_Walking2.png"));
animWalk->addSpriteFrame(spriteFrameCache->getSpriteFrameByName("EJ_Walking3.png"));
animWalk->addSpriteFrame(spriteFrameCache->getSpriteFrameByName("EJ_Walking4.png"));

animWalk->setDelayPerUnit(0.1f);
animWalk->retain();

//Animacion Saltar
animJump = Animation::create();
animJump->addSpriteFrame(spriteFrameCache->getSpriteFrameByName("EJ_Jumping1.png"));
animJump->addSpriteFrame(spriteFrameCache->getSpriteFrameByName("EJ_Jumping2.png"));

animJump->setDelayPerUnit(0.1f);
animJump->retain();

//Animacion Correr
animRun = Animation::create();
animRun->addSpriteFrame(spriteFrameCache->getSpriteFrameByName("EJ_Running1.png"));
animRun->addSpriteFrame(spriteFrameCache->getSpriteFrameByName("EJ_Running2.png"));
animRun->addSpriteFrame(spriteFrameCache->getSpriteFrameByName("EJ_Running3.png"));
animRun->addSpriteFrame(spriteFrameCache->getSpriteFrameByName("EJ_Running4.png"));

animRun->setDelayPerUnit(0.1f);
animRun->retain();

//Animacion Premuerte
animHit = Animation::create();
animHit->addSpriteFrame(spriteFrameCache->getSpriteFrameByName("EJ_Hit1.png"));
animHit->addSpriteFrame(spriteFrameCache->getSpriteFrameByName("EJ_Hit2.png"));

animHit->setDelayPerUnit(0.5f);
animHit->retain();

//Animacion Muerte
animDead = Animation::create();
animDead->addSpriteFrame(spriteFrameCache->getSpriteFrameByName("EJ_Shit1.png"));
animDead->addSpriteFrame(spriteFrameCache->getSpriteFrameByName("EJ_Shit1.png"));

animDead->setDelayPerUnit(0.5f);
animDead->retain();
```

Figura 34. Captura de la creación de las animaciones del personaje a partir de los gráficos.

Fuente: Elaboración propia.

5.2.6 Implementación de la carga de niveles.

Para la carga de niveles, primero es necesario crearlos, como se comentó en el GDD, esto lo hacemos con la herramienta Tiled. Para crear los mapas, lo mejor es usar spriteSheets, y para eso usaremos CocosStudio para agrupar nuestros gráficos creados en Photoshop en grandes hojas que los incluyan (spritesheets).

Para crear los spritesheets solo es necesario abrir CocosStudio, y dentro de él, crear un nuevo archivo (new file), y elegir SpriteSheet, donde arrastramos todas las imágenes que queramos incluir, y hacemos clic en el botón de exportar la hoja de sprites.

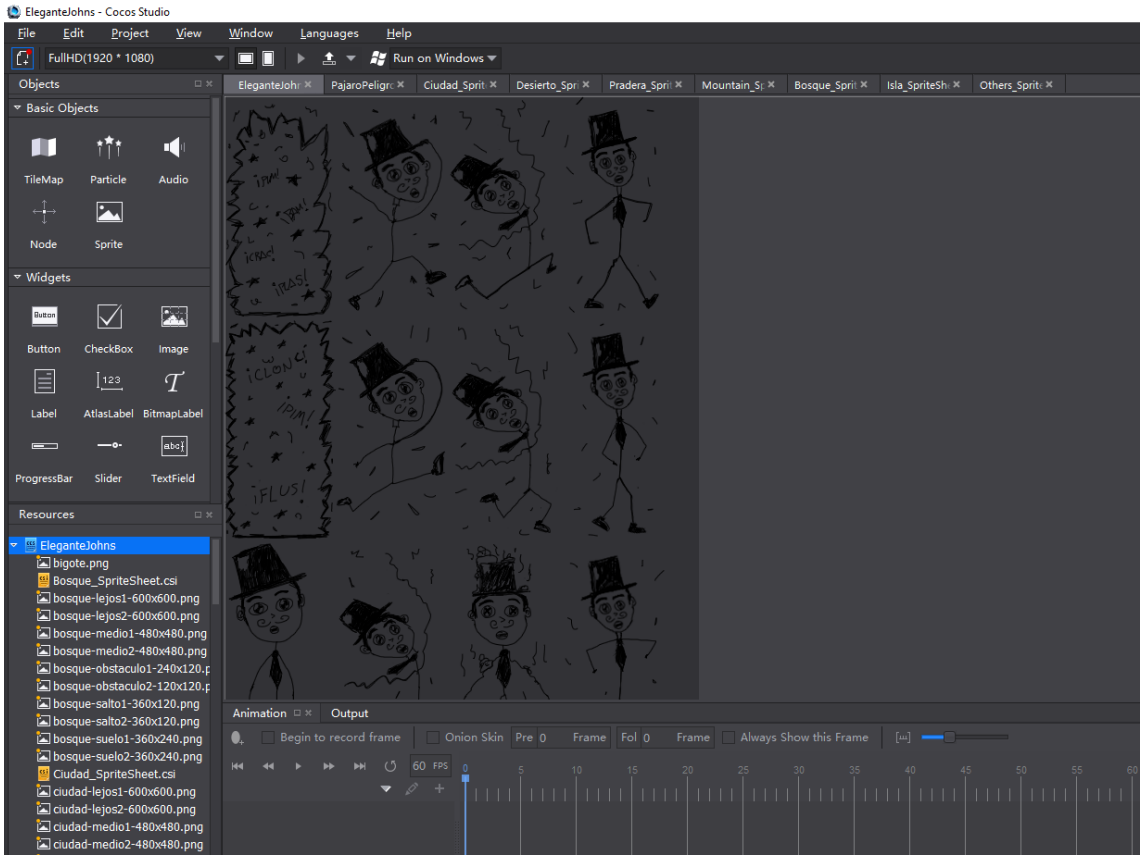


Figura 35. Captura de la elaboración de un spriteSheet con CocosStudio

Fuente: Elaboración propia.

Una vez tenemos nuestras hojas de sprites, podemos crear los niveles con Tiled. Esta herramienta nos permite crear mapas de patrones, y también grupos de objetos que luego podemos interpretar, por lo que nos viene perfecta.

Figura 36. Captura de una parte del nivel 6 en tiled.

Fuente: Elaboración propia.

Nosotros tenemos varias capas de objetos:

- Físicas. La que contiene todos los objetos de las físicas (polilíneas)
- Collectables. Contiene las monedas.
- FinNivel. Contiene la posición del fin de nivel (paraguas).
- InicioNivel. Contiene la posición del inicio del nivel.
- Peligros. Situados sobre los patrones que matarían al personaje-

Y varias capas de patrones, para cerca, medio, lejos, y adornos.

Una vez creados los mapas, nos toca interpretarlos en el proyecto mediante código. Por suerte, Cocos2d-x incluye un lector de TMX (el formato de los mapas de Tiled), así que nos ahorramos bastante trabajo, y la mayor complicación que encontramos no es en los patrones, que los cargamos fácilmente y no requieren ningún tratamiento, sino en la correcta interpretación de los objetos, aquí nadie nos puede ayudar, veamos por qué.

Cocos nos proporciona la capa de objetos, es decir, nos proporciona un objeto que contiene dentro todo lo que poseía la capa, si es una capa peligros o collectables, tiene muchos rectángulos, y si es una capa físicas, está llena de polilíneas, a su vez, estos objetos están compuestos de puntos, un rectángulo está formado por 4 puntos, y una polilínea contiene todos los puntos que le hayamos querido dar. Además aquí no queda la cosa, primero hay que interpretar todo el contenido, y luego actuar con él, indicar qué queremos hacer.

Para todo esto hemos creado una clase LevelLoader, que se encarga de primero leer todos los datos de una capa, y luego de esto, en función de lo que sea, crear un vector de objetos si se trata de collectables o peligros, crear un body de físicas que contenga todas, crear un objeto Fin Nivel, o guardarse la posición de inicio de la partida. Hecho de esta forma, solo tenemos que pedirle al LevelLoader qué queremos, y así desde Mundo, que es el contenedor del nivel, mantenemos un código mucho más limpio.

```cpp
std::vector<PolygonStructure> LevelLoader::getCoordsCapaPeligros(TMXObjectGroup *ObjectGroupLayer)
{
    //OBTENEMOS LAS CAPAS DE OBJETOS
    auto objetosCapaParsear = ObjectGroupLayer->getObjects();

    //CREAMOS UN VECTOR PARA ALMACENAR LAS POSICIONES PARSEADAS
    std::vector<PolygonStructure> VectorObjetosParseados;


    //RECORREMOS EN UN BUCLE LA CAPA DE OBJETOS OBTENIDA PARA SACAR LOS DATOS DE LAS FISICAS
    for (auto objetoParser : objetosCapaParsear)
    {
        ////CCLOG(objetito.getDescription().c_str());
        //CCLOG("****************************************");
        ////CCLOG("OBJECT CON NOMBRE: %s", objetoParser.asValueMap().at("name").asString().c_str());
        //CCLOG("****************************************");
        //CCLOG("**********   COORDS GENERALES   **********");
        //CCLOG("****************************************");
        //CCLOG("x = %f", objetoParser.asValueMap().at("x").asFloat());
        //CCLOG("y = %f", objetoParser.asValueMap().at("y").asFloat());
        //CCLOG("width = %f", objetoParser.asValueMap().at("width").asFloat());
        //CCLOG("height = %f", objetoParser.asValueMap().at("height").asFloat());


        //VARIABLES PARA ALMACENAR AQUI LAS COORDENADAS DEL HAZARD EN CUESTION
        float corX = 0;
        float corY = 0;
        float width = 0;
        float height = 0;

        //ASIGNAMOS VALORES A LAS VARIABLES DEL HAZARD
        corY = objetoParser.asValueMap().at("y").asFloat(); //Las coordenadas en y se deben invertir siempre
        corX = objetoParser.asValueMap().at("x").asFloat();
        width = objetoParser.asValueMap().at("width").asFloat(); //Las coordenadas en y se deben invertir siempre
        height = objetoParser.asValueMap().at("height").asFloat();

        //CREAMOS LA ESTRUCTURA AUXILIAR QUE CONTIENE TODO
        PolygonStructure estructuraAux;
        estructuraAux.m_positionPolygon = Vec2(corX, corY);
        estructuraAux.m_sizePolygon = Vec2(width, height);

        //METEMOS LA ESTRUCTURA EN EL VECTOR
        VectorObjetosParseados.push_back(estructuraAux);

    }
    //DEVOLVEMOS UN VECTOR DE VECTORES DE COORDENADAS CON LOS DATOS LEIDOS
    return VectorObjetosParseados;
}
```

Figura 37. Captura de cómo interpretar la capa de objetos con los peligros (hazards).

Fuente: Elaboración propia.

Y eso es todo, desde mundo solicitamos aquello que necesitamos, y lo añadimos como hijo del nodo Mundo (que a su vez es hijo de Game).

```cpp
Vector<Hazard*> * LevelLoader::getHazards(b2World * mundoFisicas)
{
    //RECOGER LA CAPA DE OBJETOS CON LOS COLLECTABLES
    TMXObjectGroup *objectosHazards = m_tiledMapHelper->getHazardsLayerObjectGroup();

    //OBTENER VECTOR DE VECTORES CON LAS COORDENADAS DE LOS OBJETOS PARSEADOS
    vectorPeligros = getCoordsCapaPeligros(objectosHazards);

    m_vectorPeligros.clear();

    //CREAMOS UNOS COLLECTABLES CON ESAS POSICIONES
    for (int i = 0; i < vectorPeligros.size(); i++)
    {
        Hazard * HazardMapa = Hazard::create();
        HazardMapa->setSize(vectorPeligros.at(i).m_sizePolygon * 0.9f); //MULTIPLICAMOS POR 0.9 EL SIZE PARA QUE SEA UN PELÍN MÁS PEQUEÑO QUE EL SPRITE

        //OJO - LA POSICION DE LOS HAZARDS LA ASIGNAMOS AQUI
        HazardMapa->setTransform(vectorPeligros.at(i).m_positionPolygon+ cocos2d::Vec2(vectorPeligros.at(i).m_sizePolygon.x/2, vectorPeligros.at(i).m_sizePolygon.y/2), 0);
        m_vectorPeligros.pushBack(HazardMapa);
    }

    return &m_vectorPeligros;
}
```

Figura 38. Captura de qué devolvemos al mundo cuando nos pide los peligros.

Fuente: Elaboración propia.

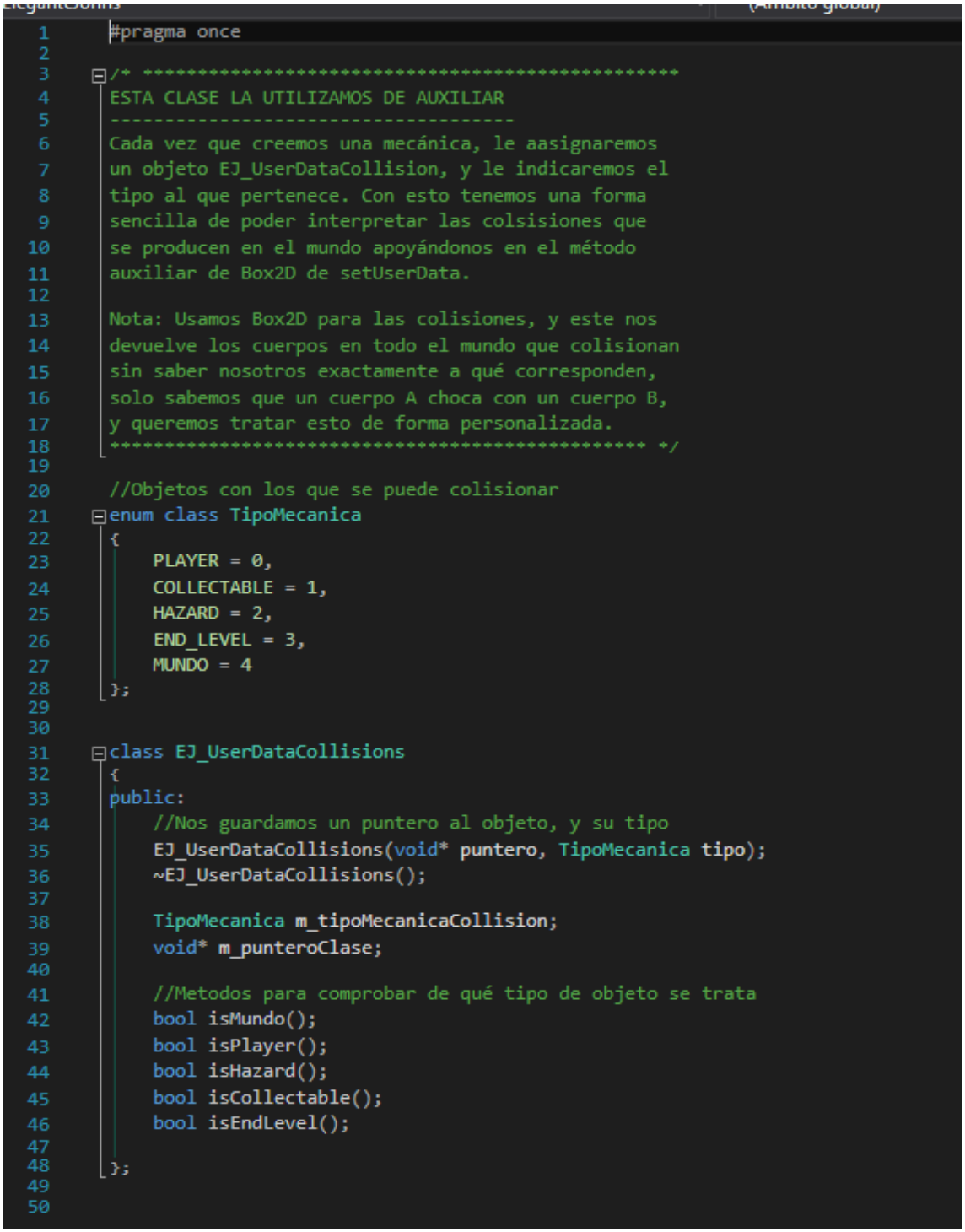
5.2.7 Implementación de las colisiones.

El tema de las colisiones lo tenemos resuelto a medias, cocos2d-x incluye el motor de físicas Box2D, y este motor nos evita calentarnos la cabeza sobre cómo implementar gravedad y otros aspectos.

Nosotros en el juego debemos encargarnos de asignar a cada elemento que esté en el nivel con manifestación física un objeto body, que introduciremos en el motor de físicas para que todas las fuerzas hagan efecto sobre ellos y puedan interactuar entre sí. Hasta aquí todo bien, y solo con esto ya el personaje choca con objetos.

La única complicación, que puede ser un pequeño quebradero de cabeza al principio, es gestionar qué choca con qué. Box2D no nos permite saber si un objeto está colisionando con algo, no podemos simplemente hacer un listener a un personaje y que todo lo que choque con él nos avise, no es tan sencillo, lo que hace este motor es que cada vez que haya una colisión en el mundo, sea lo que sea con lo que sea, nos avisa, y no sabemos qué es cada cosa.

Para solucionar este problema, box2D nos permite asignar a cada body una función lambda con lo que nosotros queramos, así que la aprovechamos. Nuestra solución ha sido crear una clase auxiliar que hemos llamado userDataCollision, que contiene un puntero que referenciará al objeto que contenga al body, y también un objeto que forma parte de un enumerado indicando el tipo de elemento que es. Añadimos además de un body, un UserDataCollision a cada objeto que tenga manifestación física en el mundo, y listo, ya tenemos interpretación de físicas.

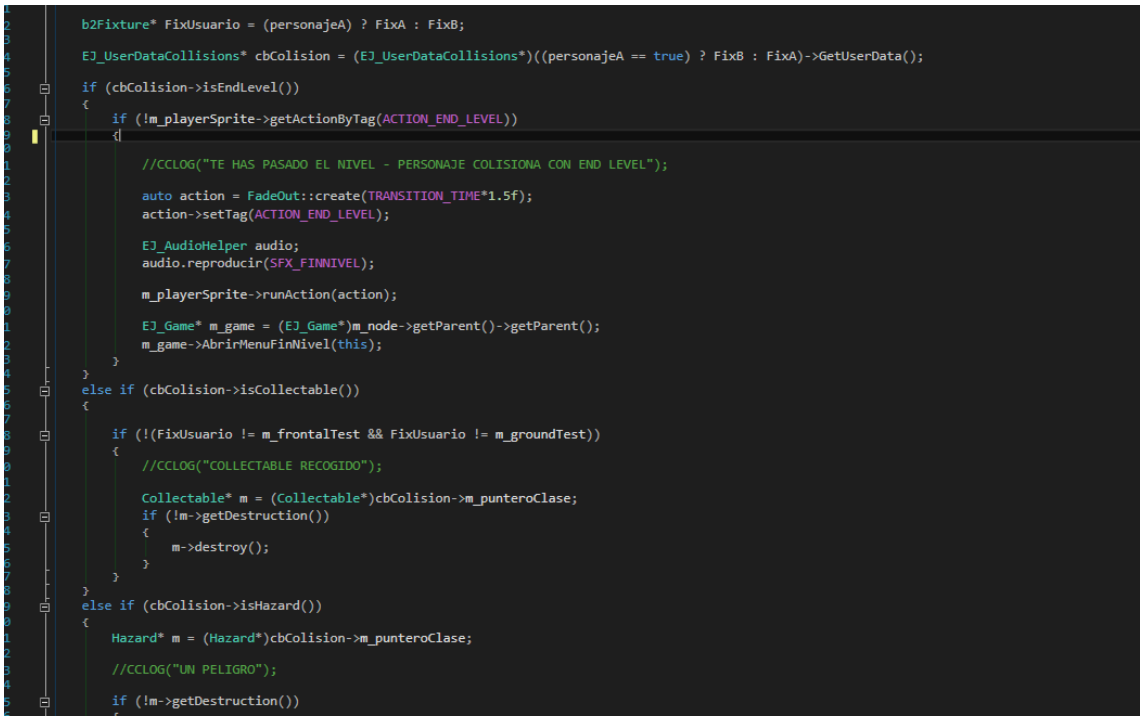


```
#pragma once

/* ***********************************************
ESTA CLASE LA UTILIZAMOS DE AUXILIAR
-------------------------------------
Cada vez que creemos una mecánica, le aasignaremos
un objeto EJ_UserDataCollision, y le indicaremos el
tipo al que pertenece. Con esto tenemos una forma
sencilla de poder interpretar las colsisiones que
se producen en el mundo apoyándonos en el método
auxiliar de Box2D de setUserData.

Nota: Usamos Box2D para las colisiones, y este nos
devuelve los cuerpos en todo el mundo que colisionan
sin saber nosotros exactamente a qué corresponden,
solo sabemos que un cuerpo A choca con un cuerpo B,
y queremos tratar esto de forma personalizada.
*********************************************** */

//Objetos con los que se puede colisionar
enum class TipoMecanica
{
    PLAYER = 0,
    COLLECTABLE = 1,
    HAZARD = 2,
    END_LEVEL = 3,
    MUNDO = 4
};


class EJ_UserDataCollisions
{
public:
    //Nos guardamos un puntero al objeto, y su tipo
    EJ_UserDataCollisions(void* puntero, TipoMecanica tipo);
    ~EJ_UserDataCollisions();

    TipoMecanica m_tipoMecanicaCollision;
    void* m_punteroClase;

    //Metodos para comprobar de qué tipo de objeto se trata
    bool isMundo();
    bool isPlayer();
    bool isHazard();
    bool isCollectable();
    bool isEndLevel();

};
```

Figura 39. Captura del .h de la clase de UserDataCollision.

Fuente: Elaboración propia

Cuando el personaje choque con algo, sabremos diferenciar fácilmente de qué se trata, y podremos tratarlo.



```
b2Fixture* FixUsuario = (personajeA) ? FixA : FixB;

EJ_UserDataCollisions* cbColision = (EJ_UserDataCollisions*)((personajeA == true) ? FixB : FixA)->GetUserData();

if (cbColision->isEndLevel())
{
    if (!m_playerSprite->getActionByTag(ACTION_END_LEVEL))
    {

        //CCLOG("TE HAS PASADO EL NIVEL - PERSONAJE COLISIONA CON END LEVEL");

        auto action = FadeOut::create(TRANSITION_TIME*1.5f);
        action->setTag(ACTION_END_LEVEL);

        EJ_AudioHelper audio;
        audio.reproducir(SFX_FINNIVEL);

        m_playerSprite->runAction(action);

        EJ_Game* m_game = (EJ_Game*)m_node->getParent()->getParent();
        m_game->AbrirMenuFinNivel(this);
    }
}
else if (cbColision->isCollectable())
{

    if (!(FixUsuario != m_frontalTest && FixUsuario != m_groundTest))
    {
        //CCLOG("COLLECTABLE RECOGIDO");

        Collectable* m = (Collectable*)cbColision->m_punteroClase;
        if (!m->getDestruction())
        {
            m->destroy();
        }
    }
}
else if (cbColision->isHazard())
{
    Hazard* m = (Hazard*)cbColision->m_punteroClase;

    //CCLOG("UN PELIGRO");

    if (!m->getDestruction())
```

Figura 40. Captura de una parte de la interpretación en personaje de las colisiones.

Fuente: Elaboración propia.

5.2.8 Implementación de la sonorización.

Tenemos la sonorización completamente resuelta con cocos2d-x, para reproducir una canción o sonido solo tenemos que llamar al motor de audio, que ya viene singleton, y desde ahí llamar a sus métodos para reproducir o parar y listo.

Se ha optado por crear una clase intermedia, a modo de helper, que nos permita gestionar mejor todo, la hemos llamado audioHelper, y contiene métodos para reproducir un sonido, y una canción, detener la canción que esté sonando, y pausar y reanudar música.

```
#include "cocos2d.h"
#include "SimpleAudioEngine.h"

/*
Creamos esta clase como helper para ayudarnos a reproducir la música y los audios
Podríamos hacerlo todo directamente usando la clase base de cocos, que además es
singleton, pero como cocos está constantemente cambiando todo, hacer esto no requería
mucho esfuerzo y lo deja todo un poco más limpio.
*/
class EJ_AudioHelper
{

public:

    //Precarga de un audio
    void preload(std::string _id);
    //Reproducir un sonido
    void reproducir(std::string _id);
    //Reproducir una canción de fondo
    void reproducirFondo(std::string _id);
    //Detener una canción (NO nos permite reanudarla)
    void parar();
    //Reanudar una canción pausada
    void continuar();
    //Pausar una canción (nos permite reanudarla)
    void pausar();
};
```

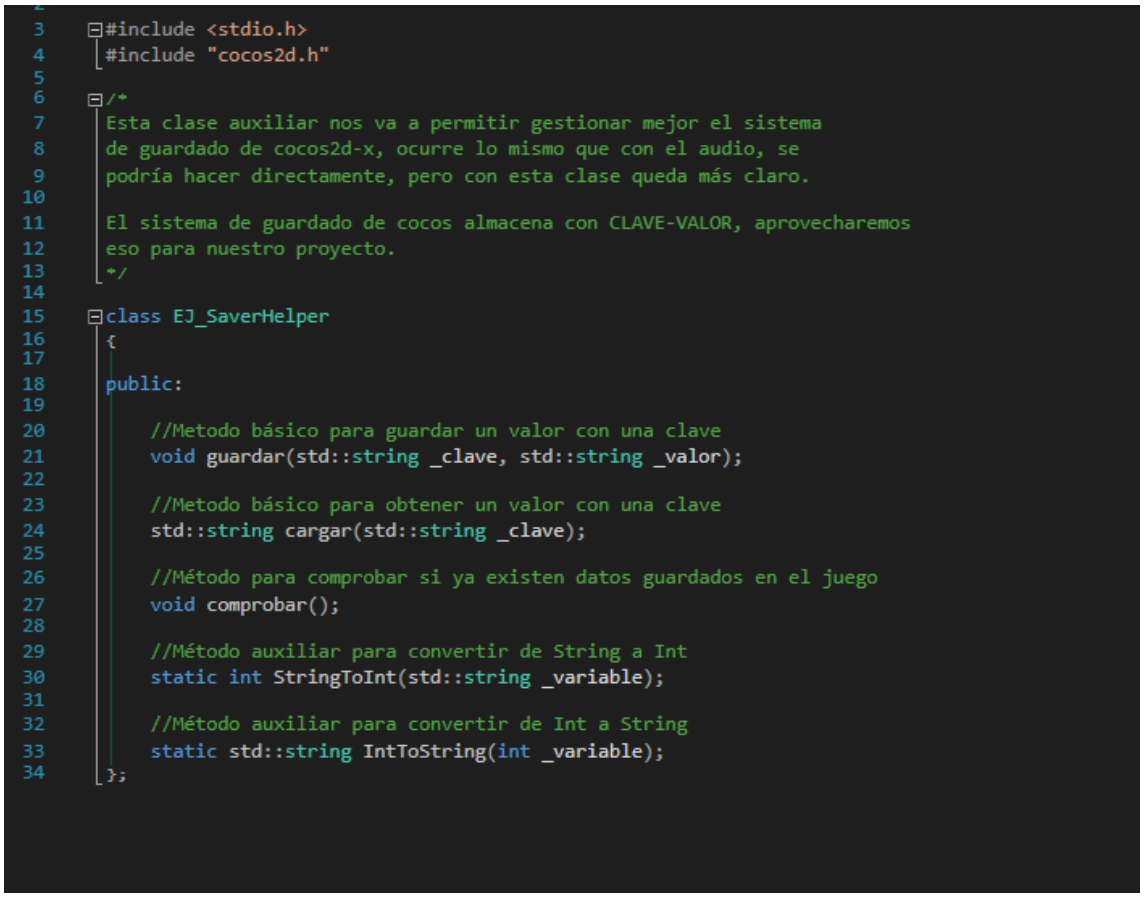
Figura 41. Captura del .h de la clase de audio helper.

Fuente: Elaboración propia.

5.2.9 Implementación del sistema de guardado

Al igual que con el tema de sonorización, el guardado lo tenemos resuelto con cocos, porque tenemos una clase UserDefault que podemos instanciar al ser singleton, y que nos permite guardar registros clave – valor, suficiente para guardar niveles superados, monedas, muertes, y otros datos sencillos que necesitemos.

También, igual que en sonorización, se ha creado una clase helper que nos ayudará a gestionar lo que guardamos y cargamos. Además, le hemos añadido un método inicializador, este método se llama al iniciarse la aplicación, y se encarga de crearnos ya todas las claves que necesitamos con un valor por defecto.



```
#include <stdio.h>
#include "cocos2d.h"

/*
Esta clase auxiliar nos va a permitir gestionar mejor el sistema
de guardado de cocos2d-x, ocurre lo mismo que con el audio, se
podría hacer directamente, pero con esta clase queda más claro.

El sistema de guardado de cocos almacena con CLAVE-VALOR, aprovecharemos
eso para nuestro proyecto.
*/

class EJ_SaverHelper
{

public:

    //Metodo básico para guardar un valor con una clave
    void guardar(std::string _clave, std::string _valor);

    //Metodo básico para obtener un valor con una clave
    std::string cargar(std::string _clave);

    //Método para comprobar si ya existen datos guardados en el juego
    void comprobar();

    //Método auxiliar para convertir de String a Int
    static int StringToInt(std::string _variable);

    //Método auxiliar para convertir de Int a String
    static std::string IntToString(int _variable);
};
```

Figura 42. Captura del .h de la clase helper del guardado y cargado de partida.

Fuente: Elaboración propia.

Por último, hemos creado un par de métodos auxiliares, que se encarguen de convertir de int a string, y de string a int, ya que guardamos todos los valores de las claves con strings.

5.2.10 Exportación a Android.

Para producir la apk de Android del proyecto tenemos que instalar una serie de programas y bibliotecas primero. Estas las podemos ver al abrir cocos, si hacemos clic en Preferences>Platform, y son las siguientes:

- Android SDK.
- Android NDK.
- JDK.
- Apache Ant.

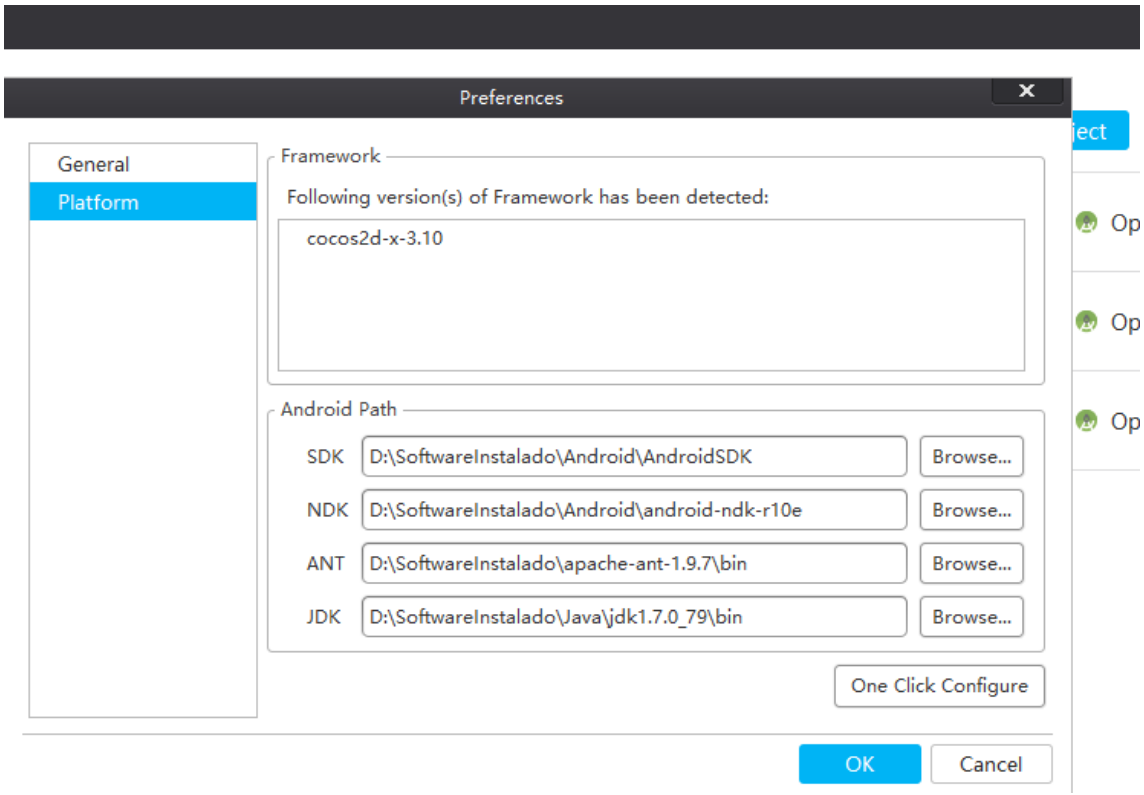


Figura 43. Captura con las rutas configuradas con lo necesario para exportar a Android.

Fuente: Elaboración propia.

Además de eso, debemos tener instalado Python en nuestro ordenador. Y configurar las variables de entorno del sistema operativo para poder acceder a todo directamente.

Luego de tener todo esto instalado y comunicado con el motor, el siguiente paso es abrir la carpeta del proyecto, y dentro de su carpeta para Android (proj-android), acceder a la carpeta jni, y ahí abrir el archivo Android.mk, en él deberemos añadir una por una todas las rutas de los .cpp del proyecto.

```
LOCAL_MODULE_FILENAME := libcocos2dcpp

LOCAL_SRC_FILES := hellocpp/main.cpp \
                   ../../Classes/AppDelegate.cpp \
                   ../../Classes/Engine2D/Base/Engine2D.cpp \
                   ../../Classes/Engine2D/Base/SimpleGame.cpp \
                   ../../Classes/Engine2D/Base/Debug/CocosDebugDraw.cpp \
                   ../../Classes/Engine2D/Base/Entidades/GameEntity.cpp \
                   ../../Classes/Engine2D/Base/Entidades/PhysicsGameEntity.cpp \
                   ../../Classes/Engine2D/Control/VirtualControls.cpp \
                   ../../Classes/Engine2D/Control/ControlElegante.cpp \
                   ../../Classes/Engine2D/Level_Loader/LevelLoader.cpp \
                   ../../Classes/Engine2D/Level_Loader/TiledMapHelper.cpp \
                   ../../Classes/Escenas/Menus/EJ_CreditosMenu.cpp \
                   ../../Classes/Escenas/Menus/EJ_ElegirNivelMenu.cpp \
                   ../../Classes/Escenas/Menus/EJ_MenuPrincipal.cpp \
                   ../../Classes/Escenas/Menus/EJ_SplashScreen.cpp \
                   ../../Classes/Escenas/Niveles/Game/EJ_Game.cpp \
                   ../../Classes/Escenas/Niveles/Level/Mundo.cpp \
                   ../../Classes/Escenas/UI/GUI.cpp \
                   ../../Classes/GameSharing/GameSharing.cpp \
                   ../../Classes/Mecanicas/EJ_Mechanic.cpp \
                   ../../Classes/Mecanicas/Collectable.cpp \
                   ../../Classes/Mecanicas/EndLevel.cpp \
                   ../../Classes/Mecanicas/Hazard.cpp \
                   ../../Classes/Personaje/EJ_Player.cpp \
                   ../../Classes/Utils/EJ_AudioHelper.cpp \
                   ../../Classes/Utils/EJ_SaverHelper.cpp \
                   ../../Classes/Utils/EJ_UserDataCollisions.cpp
LOCAL_C_INCLUDES := $(LOCAL_PATH)/../../Classes

# _COCOS_HEADER_ANDROID_BEGIN
# _COCOS_HEADER_ANDROID_END
```

Figura 44. Captura del Android.mk con los .cpp añadidos.

Fuente: Elaboración propia.

Finalmente, abrir la consola de comandos, llegar hasta la ruta del proyecto, y en ella ejecutar el comando “cocos compile -p android”, o añadirle a lo anterior “-m release” si queremos la versión firmada que podamos subir a google play.

```
-release-sign:
    [echo] Signing final apk...
 [zipalign] Running zip align on final apk...
    [echo] Release Package: D:\workspace\Cocos\EleganteJohns\proj.android\bin\EleganteJohns-release.apk
[propertyfile] Creating new property file: D:\workspace\Cocos\EleganteJohns\proj.android\bin\build.prop
[propertyfile] Updating property file: D:\workspace\Cocos\EleganteJohns\proj.android\bin\build.prop
[propertyfile] Updating property file: D:\workspace\Cocos\EleganteJohns\proj.android\bin\build.prop
[propertyfile] Updating property file: D:\workspace\Cocos\EleganteJohns\proj.android\bin\build.prop

-post-build:

release:

BUILD SUCCESSFUL
Total time: 15 seconds
Move apk to D:\workspace\Cocos\EleganteJohns\bin\release\android
Build succeed.
D:\workspace\Cocos\EleganteJohns>
```

Figura 45. Captura del resultado final del terminal tras pedirle que genere la apk de Android.

5.2.11 Implementación de los logros.

Para implementar los logros utilizamos la extensión de cocos GameSharing[23], que nos permite una comunicación sencilla con los Google Play Services, para introducir y desbloquear logros,

abrir panel de logros, y de marcadores. Además, debemos subir la apk a Google Play para poder gestionarla.

Lo anterior lo hacemos descargando la extensión desde el github del autor, y copiando e incluyendo todos los archivos en el proyecto, además de copiar todas las clases java que incluye donde corresponde dentro de la carpeta android, recursos, y la biblioteca de Google Play Services.

Una vez hecho esto, solo es necesario incluir en una clase a GameSharing, y usar sus métodos. Para añadir los logros, debemos hacerlo primero desde Google Play, y al hacerlo, cada logro tiene un identificador y una identificación cifrada, esto lo copiamos y pegamos en un documento de los resources de Android que hemos pegado al principio para introducir GameSharing, en concreto, en el documento ids.xml.

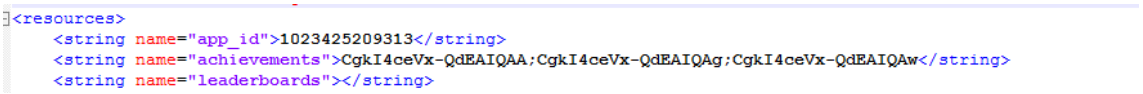


Figura 46. Cómo incluir los logros en el documento ids.xml

Fuente: Elaboración propia.

Realizados los pasos anteriores, nos creamos un enumerado con orden creciente empezando por cero, y cuando llamemos al método de GameSharing para desbloquear logros, al que hay que pasarle un int de id, los logros corresponderán al orden como los hayamos añadido en nuestro documento de ids.

6 Conclusiones

Una vez realizado el proyecto, donde hemos podido realizar en mayor o menor medida los objetivos planteados. Contamos con un videojuego para Android, sencillo pero con varios niveles, con gráficos propios, que hace uso de logros, y realizado con cocos2d-x, y con todos los objetivos cumplidos, podemos concluir en que hemos obtenido un resultado correcto.

Como observaciones acerca del motor, es evidente que nos encontramos ante una herramienta bastante buena para producir videojuegos, pero a la que todavía le falta bastante maduración.

Mientras que con código el desarrollo del proyecto se puede llevar, para el apartado gráfico, cocos studio no ha servido para nada más que para crear hojas de sprites, ofrece muchas funcionalidades, pero son demasiado complicadas de utilizar, con fallos muy diversos, sobre todo para la creación de menús, y no existen guías adecuadas para poder utilizarlo.

Aparte de esto, no está correctamente mantenido, en la versión 3.10 que hemos utilizado, el Android NDK que soportaba debía ser una versión vieja necesariamente, y darse cuenta de eso es problemático.

Además, el motor no proporciona herramientas para depurar el código en Android. Nosotros hemos desarrollado el proyecto en Windows, y luego hemos exportado, por lo que muchas cosas podían fallar (y fallaron), para averiguar por qué la aplicación no funcionaba era una odisea y no bastaba con simplemente sacar la consola de Android Studio o en terminal. Al final era necesario ir probando poco a poco qué podía estar fallando y guiarse de la intuición.

Otro problema del motor Cocos es que no podemos crear proyectos con las bibliotecas precompiladas en el caso de Windows, si lo intentamos, aunque el proyecto se genera, es imposible ejecutarlo, por lo que solo es posible hacerlo con todo el código fuente y compilarlo manualmente (tardando unos 15 minutos aproximadamente), y para hacer pruebas rápidas, esto es algo muy pesado.

Por último, respecto a las observacioens negativas, para poder hacer debug en el proyecto, se utilizaba el método de cocos CCLOG, algo interno de cocos que no tiene que dar ningún problema, sin embargo, antes de exportar a Android, aunque sea para pruebas, necesitamos comentarlos o quitarlos, porque si no, puede dar problemas. Y no solo eso, también algunas funciones de std no están disponibles si queremos exportar a Android, pero eso no lo sabemos hasta que no intentemos exportar, y es entonces cuando nos damos cuenta que debemos hacer muchas correcciones.

Como punto a favor, el proyecto, aunque era sencillo, en cuanto a clases acabó teniendo una cantidad considerable. Así que nos obliga a mantener el código lo mejor organizado posible, y

comentarlo era esencial durante todo el desarrollo para evitarse muchos quebraderos de cabeza extras.

Podemos añadir también a los puntos positivos que con este motor trabajamos con una resolución de diseño, esto quiere decir que no es necesario preocuparse especialmente por qué cálculos hacer en función de la parte visible de la pantalla, simplemente asignamos posiciones absolutas en función de esta resolución de diseño, y automáticamente se adaptará él solo en función del terminal.

Para acabar con los detalles positivos, está bien que el motor tenga una integración tanto de un motor de físicas, como lector de mapas TMX, nos ahorra trabajo de integración de bibliotecas externas y posibles problemas que estas llegaran a ofrecer.

Con todo lo anterior dicho, y teniendo experiencia en haber desarrollado en otros motores, como Unreal Engine 4 o Unity, la verdad es que compensa muchísimo utilizarlos, porque obtener una calidad superior resulta mucho más sencillo y sin complicarse tanto al tener una comunidad de usuarios mucho mayor, y un mayor presupuesto al pedir royalties o cobrar licencias especiales. Quizá en unos años cocos sea muy recomendable si corrigen todos los problemas descritos, pero mientras no sea así, sería recomendable explorar otras opciones.

6.1 Propuestas de mejora y trabajo futuro

Como propuestas de mejora y trabajo futuro se contempla, entre otros aspectos, añadir animaciones a muchos componentes estáticos del menú y mejorar algunas interfaces para otorgarle mucha más vida al videojeugo.

También se contempla añadir mayor dificultad a los niveles para que se tarde más en terminarlo.

Finalmente, sería conveniente añadir más logros para atraer más a la rejugabiliad.

7 Bibliografía y referencias

7.1 Introducción

[1] Historia de los videojuegos:

https://es.wikipedia.org/wiki/Historia_de_los_videojuegos#Balance.2C_presente_y_futuro_de_los_videojuegos

[2] Facturación de los videojuegos:

http://elpais.com/diario/2008/04/09/radiotv/1207692005_850215.html

[3] Pokémon Go Éxito:

http://www.t13.cl/noticia/tendencias/el-exito-financiero-pokemon-go-ha-recaudado-mas-us-440-millones-mundo

[4] Samsung Gear videojuegos:

http://www.xataka.com/realidad-virtual-aumentada/los-19-juegos-y-aplicaciones-imprescindibles-para-las-gafas-gear-vr

[5] Flappy Bird éxito:

https://es.wikipedia.org/wiki/Flappy_Bird

[6] Motor Cocos:

http://www.cocos2d-x.org/

7.2 Marco Teórico o Estado del Arte

[7] Definición de videojuego por Bernard Suits:

The Grasshopper: Games, Life and Utopia, Bernard Suits, 1978.

[8] Definición de videojuego por Wikipedia:

https://es.wikipedia.org/wiki/Videojuego

[9] Historia de los motores

Game Engine Architecture, Jason Gregory, 2015 (second edition)

[10] Unreal Engine se crea en 1998:

https://es.wikipedia.org/wiki/Unreal_Engine

[11] El motor Source se presenta en 2004:

https://es.wikipedia.org/wiki/Source

[12] Lista de motores de videojuegos:

https://en.wikipedia.org/wiki/List_of_game_engines

[13] Logo Cocos2d-x:

http://www.cocos2d-x.org/attachments/download/801

[14] Chukong Technologies mantiene Cocos2d-x:

https://en.wikipedia.org/wiki/Chukong_Technologies#Cocos_2D-x_Game_Engine

[15] Cocos Creator:

http://cocos2d-x.org/docs/editors_and_tools/creator/

[16] Logo Unity:

https://upload.wikimedia.org/wikipedia/commons/5/55/Unity3D_Logo.jpg

[17] Consejos elección de motor en diapositivas de la asignatura "Motores para videojuegos" de Videojuegos 2.

https://moodle2014-15.ua.es/moodle/pluginfile.php/18327/mod_resource/content/4/vii-08-motores.pdf

[18] Diagrama Arquitectura Cocos2d-x:

https://mastermoviles.gitbooks.io/videojuegos-moviles/content/imagenes/juegos/cocos2d_scene2d.jpg

7.3 Metodología

[19] Metodología Kanban:

http://www.javiergarzas.com/2011/11/kanban.html

7.4 Diseño del Videojuego

[20] Logo Pegi3:

https://upload.wikimedia.org/wikipedia/commons/thumb/2/2c/PEGI_3.svg/426px-PEGI_3.svg.png

[21] Información Pegi:

http://www.pegi.info/es/index/id/96/

[22] Definición RAE:

http://dle.rae.es/?id=MaawoiZ

7.5 Desarrollo e implementación

[23] GameSharing:

https://github.com/dwd31415/GameSharing

7.6 Bibliografías frecuentemente visitadas

Mi memoria de TFG “Desarrollo de un videojuego en Unreal Engine 4”:

https://rua.ua.es/dspace/bitstream/10045/49409/1/Desarrollo_de_un_videojuego_con_Unreal_Engine_4_EGEA_CANALES_JOSE_MARIA.pdf

La web de documentación de cocos2d-x:

http://www.cocos2d-x.org/docs/api-ref/cplusplus/v3x/